DOCTOR OF SCIENCES

On Multiplicities in Coordination Languages

Darquennes, Denis

Award date:
2017

Awarding institution:
University of Namur

On Multiplicities in Coordination Languages

Denis Darquennes

Faculty of Computer Science

University of Namur

Thesis submitted for the degree of

Doctor of Science,
option Computer Science

Namur, December 7th 2017

Jury

Prof. Erik De Vink
Eindhoven University of Technology, Netherland

Prof. Vincent Englebert
University of Namur, Belgium

Prof. Jean-Marie Jacquet
University of Namur, Belgium

Prof. Isabelle Linden
University of Namur, Belgium

Dr. Gilles Perrouin
University of Namur, Belgium

Prof. Wim Vanhoof
University of Namur, Belgium

Prof. Denis Zampunieris
University of Luxemburg, Luxemburg

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Science in the subject of Computer Science

Supervised by Prof. Jean-Marie Jacquet and Prof. Isabelle Linden

University of Namur
FOCUS Research Group

Abstract

Coordination languages provide a nice framework for separating computations from interaction among components. Promoted by Carriero and Gelernter, Linda is the first coordination language having been proposed. Since then, many variants have been studied in the coordination community, among others by the Coordination research group at the University of Namur. Most of them however suffer from the Linda property of manipulating only one tuple at a time, which is obviously not expressive enough to tackle service oriented applications where, for instance, recommendation is a key feature.

In this context, this thesis proposes and studies several extensions of Bach, a Linda dialect developped at the University of Namur. They offer the possibility of manipulating many instances of a same token at a time as well as of manipulating simultaneously many instances of different tokens. Our work includes both the definition of the languages, the study of their expressiveness, the design of implementations as well as the conception of tools for reasoning on programs written with them.

A mes parents, Denise et Henri,
pour la vie qu'ils m'ont donnée
et l'amour dont ils m'ont comblé.

Acknowledgements

I would like to thank my two supervisors Professor Jean-Marie Jacquet and Professor Isabelle Linden. With their help I discovered the world of coordination languages, and they were a great example of scientific rigour during all the years of my work. I thank them for their support, expertise and encouragement at every step of my work, and especially during the writing of the scientific articles. I thank more specifically Jean-Marie for his presence in the conferences where I communicated my results, and for his introduction to the Professor Antonio Brogi, Professor Erik de Vink, Professor Farhad Arbab and Professor José Proença. I thank all of them for their help and answers to some of my requests during my research.

I express my gratitude to the members of the accompanying committee for their interest in the content of the thesis, and for their recommendation near the Faculty of Computer Science of the University of Namur, to provide me additional time to finish my work. I thank the authorities of the Faculty of Computer Science and of the University for their agreement.

I express also my gratitude to the members of the jury, Professors Isabelle Linden, Vincent Englebert, Erik De Vink, Denis Zampunieris, Gilles Perrouin, Wim Vanhoof and Jean-Marie Jacquet for their evaluation of my work and their comments on earlier drafts of it.

I thank the Professors of the Faculty of Computer Science, from which I have been the teaching assistant, with a special thought to Professor Jean-Paul Leclercq, Professor Marie-Ange Remiche and Professor Jean-Marie Jacquet. I greatly appreciated to collaborate with them. I specifically thank Jean-Marie for his trust in me, and our perfect collaboration. I admired his constant enthusiasm and pedagogical perfection. I thank all my students, for their participation during the courses. This has always been my best return.

I have a special thought to all the teachers and people I met in my life, and that have contributed to make me progress, and to become – I hope – more Human. Among them I thank my wife Claudette Kibasha for the life we share now together,

and for her perspective on the world, and I thank my teachers of the first hour, my parents Henri and Denise, that have always supported me, in every project that I have undertaken. With all my love to them.

December 2017 Denis Darquennes

Contents

Part I

Background

Chapter 1

Introduction

1.1 The current context

1.1.1 Density in coordination languages

The technological evolution over the last recent years confirm the upward trends in pervading our everyday environments with new mobile devices injecting or retrieving information from very dynamic and dense networks.

Internet falls within this scope. But as explained in [BCGZ01], "the Internet is today much more than a mere distributed information repository, or a world-wide collection of network services. Instead it constitutes a global, distributed, open, heterogeneous, descentralised, and unpredictable environment". It is clear that, in such an environment, coordination plays an essential role, not only at the technological level, but also at the application level.

The existing languages for coordination and in particular those based on the exchange of tuples through a shared dataspace offer an elegant response to such a constraint. Among them, the Bach language – a dialect of Linda developed at the University of Namur – permits to model in an elegant way the interaction between different components through the deposit and retrieval of tuples in a shared space. However, together with other coordination languages it suffers from several limits. The manipulation of one tuple at a time, that induces a non-deterministic choice between several tuples matching a required one, is one of them. Moreover, as service-oriented applications become more and more available, the rapid evolution of their demand induces a competition between them, requiring a vast adaptive capacity. The ability to measure their popularity and quality of services is then crucial for their evolution, as well as their survival on the

Internet. Again with respect to that point Linda-like languages do not offer proper mechanisms.

It appears that those limits can be circumvented by providing an atomic global selection of a finite number of tokens. We obtain this property by considering the level of presence of the tokens in the shared space, a level that we call density. Many applications can take advantage of the densities associated with information, represented by the tokens. The intuition is that the more frequent a tuple is present on the tuplespace, the more likely it will be discovered to provide an answer to a question or quality request. Moreover, requiring tuples beyond a minimum level of density guarantees that only those with a sufficient recognition will be selected. As we shall see in the next subsections, many examples can be treated by this new extension of the Linda language, and in many different contexts.

1.1.2 The taxi application and the need for domain specific coordination languages

In a commercial context, we may easily develop applications allowing to select some actors on the basis of a specific criteria. For instance, the actors could be taxi drivers, with their reputation in quality of service as criteria of selection. To make it operational, the system needs on the one hand to allow users to express their satisfaction with regard to the service provided, and on the other hand, to test that a taxi driver is recognized at a sufficient level of satisfaction. For the following, we will assume that only positive marks are taken into account and that the service offered by a taxi driver can be evaluated as good or excellent, corresponding to a respective evaluation with number 1 or 2. We will then imagine that a level of satisfaction 100 is a minimal satisfaction mark for a reasonable driver. As we shall see later, other similar applications can be imagined in the fields of security or for smart traffic management.

To illustrate our claim for a domain specific coordination language, let us first show how easy it is to program the taxi application by using the Bach language, extended with density. Then we present a way to code it in java, using threads to provide concurrency, and thereafter we compare the two approaches.

For the first task, the satisfaction of a user can be registered by inserting the tuple `<taxi_driver_id>` once if the evaluation mark is good and twice if it is excellent. Technically, with `taxi_driver_id` being the identifier of the taxi driver, this amounts to respectively executing `tell(<taxi_driver_id>(1))` or `tell(<taxi_driver_id>(2))`. As regards the second task, making sure that a proposed driver, say identified by `id`, has reached a level of satisfaction of at least 100, can be simulated by executing the primitive `ask(<id>(100))`. Note

that, as the number of matching tuples is only counted, such a satisfaction level may be reached thanks to the contribution of many users. Of course, different policies can be implemented in the application, for instance to forbid a user to mark a taxi driver more than once a day. It is also worth noting that thanks to the space and time decoupling between information producers and information consumers offered by coordination languages, it is very easy to introduce new users and new taxi drivers in the application. This kind of application could also be used for other businesses, like restaurants, hairdressers, plumbers, electricians or building contractors.

To further illustrate the coding in Bach, we present hereafter some screenshots of the application running with the command line simulators discussed in Chapter 8. The client has two possibilities. The first consists in choosing to evaluate the quality of service of a driver, supposing for instance that the driver has an actual evaluation of 125.

```
 Welcome to Dense Bach version 1.
 Type in agents to evaluate them.

DBach> tell(cab01(2)).
DBach> >> Request 1 launched

DBach> >> tell(cab01(2)) successfully terminated
       >> store : { cab01(127) }

DBach> >> Request 1 successfully terminated
```

The second choice of the client is to know if the actual popularity of its driver is higher than 100, supposing the previous evaluation of 127. The answer shows the actual value and confirm a successful answer to the request.

```
 Welcome to Dense Bach version 1.
 Type in agents to evaluate them.

DBach> tell(cab01(2)).
DBach> >> Request 1 launched

DBach> >> tell(cab01(2)) successfully terminated
       >> store : { cab01(127) }

DBach> >> Request 1 successfully terminated

DBach> ask(cab01(100)).
DBach> >> Request 2 launched

DBach> >> ask(cab01(100)) successfully terminated
       >> store : { cab01(127) }

DBach> >> Request 2 successfully terminated
```

For comparison, we now present the java code for the same application. It consists in two main parts written in two files. The first one concerns the server of the taxi company, that manages the reputation level of every taxi driver working for it. The second file is in charge of managing the requests of clients to the server. An instance of this class is created for every new connected agent.

The first file *Server.java* is composed of two classes. The first one is the public *Server* class that contains the main method. This class has a variable *ServerSocket* to connect the clients to the server. The main method contains a mapping for storing the identifications of the taxi cabs, represented by a string, and their associated popularities, represented by an integer. This class opens a socket on a specific port and announces that the server listens on it. Any accepted client is connected through a specific socket. The server creates a specific thread for any new client, by invoking the second class *TaxiServerThread*. This class captures any incoming flux from the client, starting by getting the identifier of the taxi cab. As long as the client does not signal the end of the communication with a specific string *bye*, the server reacts to the only two possible actions : on the one hand *consult* and on the other hand *give*. With the first one the client wants to obtain the actual popularity of the taxi cab. The server provides it unless in case of a new driver, for which the popularity is not yet evaluated. With the second action the client communicates to the server its evaluation of the service of the taxi cab. The server registers the evaluation and confirms it to the client. Figures 1.1 to 1.3 list the code of these two *server* classes.

```
import java.io.*;
import java.net.*;
import java.util.HashMap;
import java.util.NoSuchElementException;
import java.util.Scanner;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

public class Server {

 public static ServerSocket ss = null;

 public static void main(String[] args) {

        HashMap<String, Integer> hm = new HashMap<String, Integer>();
        int port = 2009;

        try {

            ss = new ServerSocket(port);
            System.out.println("Taxi server is listening on port "+ port);

            while(true){

                  Socket s = ss.accept();
                  System.out.println("Connection established to a client");

                  Thread t = new Thread(new TaxiServerThread(s,hm));
                  t.start();
            }

        } catch (IOException e) {
                  System.out.println("System exception for port "+port);
        }
    }
}
```

Figure 1.1: The server class (1)

```
  class TaxiServerThread implements Runnable {

        private HashMap<String, Integer> hm;
        private Socket s;

        private String cabId = "unknown";
        public boolean endOfSession = false;

        public TaxiServerThread(Socket s, HashMap<String, Integer> hm) {
         this.s = s;
         this.hm = hm;
        }

        public void run() {

        String cabId;
        String input;
        int popularity;
        int count;

        String client = s.getInetAddress().toString();
        System.out.println("Connected to "+ client);

        try {    ... see following figure

            } catch (Exception e) {
            e.printStackTrace();
        }

        System.out.println("Closed connection to " + client);
    }
}
```

Figure 1.2: The server class (2)

```
try {

    Scanner in = new Scanner(s.getInputStream());
    PrintWriter out = new PrintWriter(s.getOutputStream(),true);

    System.out.println("Welcome to Taxi Server\n");

    cabId = in.nextLine(); // get cab id

    while (!endOfSession) {

        if (cabId.equalsIgnoreCase("bye")) {

            endOfSession = true;

        } else {

            input = in.nextLine(); // get Action
            if (input.equalsIgnoreCase("consult")) {
                  if(hm.containsKey(cabId)) {
                        popularity = hm.get(cabId);
                        out.println("Popularity is : "+popularity);
                  } else {
                        out.println("New driver : popularity not yet evaluated.");
                  }
            } else {
                 if (input.equalsIgnoreCase("give")) {

                     input = in.nextLine(); // get score
                     if(hm.containsKey(cabId)) {
                            count = hm.get(cabId);
                            count = count + Integer.parseInt(input);
                            hm.put(cabId,count);
                            out.println("Evaluation registered, thank you.");
                     } else {
                            count = Integer.parseInt(input);
                            hm.put(cabId,count);
                            out.println("Evaluation registered, thank you.");
                     }
                 } else {
                     if (input.equalsIgnoreCase("bye")) {
                         endOfSession = true;
                     } else {
                         out.println("Entry error");
                     }
                 }

            }
        }
    }

    s.close();
```

Figure 1.3: The server class (3)

The second file *Client.java* contains the class that describes the behaviour of any client of a taxi cab, and that connects to the server. The main method of the class first prints the different possible actions provided to a client: *consult*, *give* and *bye*. It also asks the client to provide the Internet adress of the server he wants to get into contact. This connection is performed by a private method *getSocket*, that returns the socket in charge of the connection. The client has then to provide the identifier of its taxi cab. With this piece of information as long as the client does not signal the end of the session, he has the choice between asking to obtain the actual popularity of the taxi driver, or to give its score about the quality of service. Figures 1.4 to 1.6 list the code of the *client* class.

```
import java.io.*;
import java.net.*;
import java.util.Scanner;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.BufferedReader;

public class Client {

    public static void main(String[] args) {

        int port = 2009;
        boolean endOfSession = false;

        String command;
        String popularityMessage;
        String score;
        String registrationMessage;
        String errorMessage;

        System.out.println("Welcome to the Taxi Client\n");
        System.out.println("Type ");
        System.out.println("  - consult to consult the popularity of a taxi ");
        System.out.println("  - give to provide the popularity of a taxi (2 excellent , 1 good)")
        System.out.println("  - bye to leave the service\n");


        try { ...  see following figure

        } catch (Exception e) {
                e.printStackTrace();
        }

    }
```

Figure 1.4: The client class (1)

```
try {

    Socket s = getSocket(port);

    System.out.println("Connected on port " + port);

    Scanner in = new Scanner(s.getInputStream());
    PrintWriter out = new PrintWriter(s.getOutputStream(),true);
    Scanner inCli = new Scanner(System.in);

    System.out.print("\nEnter the cab id : ");
    String cabId = inCli.nextLine();
    out.println(cabId);

    while (!endOfSession) {

      if (cabId.equalsIgnoreCase("bye")) {

          endOfSession = true;

      } else {

          System.out.print("Command : ");
          command = inCli.nextLine();
          out.println(command);

     if (command.equalsIgnoreCase("consult")) {
                popularityMessage = in.nextLine();
                System.out.println(popularityMessage);
          } else {

               if (command.equalsIgnoreCase("give")) {
                    System.out.print("Provide score : ");
                    score = inCli.nextLine();
                    out.println(score);
                    registrationMessage = in.nextLine();
                    System.out.println(registrationMessage);
               } else {
                     if (command.equalsIgnoreCase("bye")) {
                          endOfSession = true;
                     } else {
                          errorMessage = in.nextLine();
                          System.out.println(errorMessage);
                     }
               }
          }

      }

    }

  s.close();
}
```

Figure 1.5: The client class (2)

```
    private static Socket getSocket(int port) {

        Socket s;
        String host;
        InetAddress ip;

        Scanner sc = new Scanner(System.in);

        while (true) {

            System.out.print("To which Taxi Server do you want to connect? ");
            host = sc.nextLine();

            try {

                ip = InetAddress.getByName(host);
                s = new Socket(ip,port);
                return s;
            }
            catch (UnknownHostException e) {
                System.out.println("The host is unknown");
            }
            catch (IOException e) {
                System.out.println("Network error");
            }

        }

    }
}
```

Figure 1.6: The client class (3)

As for the implementation in Bach, we present hereafter screenshots of the execution of the java program. The first screenshot shows the connection established between the taxi server and a client, on a specific port 2009.

```
Taxi server is listening on port 2009
Connection established to a client
Connected to /127.0.0.1
Welcome to Taxi Server
```

The second screenshot shows the informations provided by the client : the Internet adress of the server (on localhost) and the identifier of the taxi cab, in this case *cab01*.

```
Welcome to the Taxi Client

Type
  - consult to consult the popularity of a taxi
  - give to provide the popularity of a taxi (2 excellent, 1 good)
  - bye to leave the service

To which Taxi Server do you want to connect ? localhost
Connected on port 2009

Enter the cab id : cab01
```

Client and server being now connected, the client has now to choose between the different commands : *consult*, *give* and *bye*.

```
Welcome to the Taxi Client

Type
  - consult to consult the popularity of a taxi
  - give to provide the popularity of a taxi (2 excellent, 1 good)
  - bye to leave the service

To which Taxi Server do you want to connect ? localhost
Connected on port 2009

Enter the cab id : cab01
Command :
```

If the client chooses to first consult the popularity of its taxi cab, he types in *consult*. As the driver has not yet been evaluated, the answer of the server specifies it.

```
Welcome to the Taxi Client

Type
  - consult to consult the popularity of a taxi
  - give to provide the popularity of a taxi (2 excellent, 1 good)
  - bye to leave the service

To which Taxi Server do you want to connect ? localhost
Connected on port 2009

Enter the cab id : cab01
Command : consult
New driver : popularity not yet evaluated.
Command :
```

Now the client chooses to give its appreciation, by typing in *give*. He is then asked to provide a score, say for instance 2. When acted, the score is registered, and confirmed by the server.

```
Welcome to the Taxi Client

Type
  - consult to consult the popularity of a taxi
  - give to provide the popularity of a taxi (2 excellent, 1 good)
  - bye to leave the service

To which Taxi Server do you want to connect ? localhost
Connected on port 2009

Enter the cab id : cab01
Command : consult
New driver : popularity not yet evaluated.
Command : give
Provide score : 2
Evaluation registered, thank you.
Command :
```

If the client wants to check again the global popularity of its taxi cab, he types again *consult*, with an answer now equal to 2.

```
Welcome to the Taxi Client

Type
  - consult to consult the popularity of a taxi
  - give to provide the popularity of a taxi (2 excellent, 1 good)
  - bye to leave the service

To which Taxi Server do you want to connect ? localhost
Connected on port 2009

Enter the cab id : cab01
Command : consult
New driver : popularity not yet evaluated.
Command : give
Provide score : 2
Evaluation registered, thank you.
Command : consult
Popularity is : 2
Command :
```

Finally the client signals to the server the end of the communication, with the *bye* command. From his part, the server prints the closure of the connection.

```
Welcome to the Taxi Client

Type
  - consult to consult the popularity of a taxi
  - give to provide the popularity of a taxi (2 excellent, 1 good)
  - bye to leave the service

To which Taxi Server do you want to connect ? localhost
Connected on port 2009

Enter the cab id : cab01
Command : consult
New driver : popularity not yet evaluated.
Command : give
Provide score : 2
Evaluation registered, thank you.
Command : consult
Popularity is : 2
Command : bye
```

```
Connection established to a client
Connected to /127.0.0.1
Welcome to Taxi Server

Closed connection to /127.0.0.1
```

As the reader will immediately notice, the code to deploy the taxi application is much smaller with our approach than by using Java. This is essentially due to two factors. On the one hand, the shared space offered by coordination languages like Bach dispenses the programmer from writing explicitly a server and from handling the exchange of messages. In particular, information is searched on the basis of its contents and not through a predefined protocol of messages being exchanged. On the other hand, concurrency is provided directly without the need to cope for threads.

1.1.3 Other applications

Other applications can be coded similarly to the taxi application. In the security field, many potentially critical situations can be managed by information systems coordinated with dynamic centralised data centres. Examples of such situations concern the emergence of saturation of crowds, in metro stations, in shopping malls, in airports or even in cultural or sports events. In all these situations, one could imagine to count on the one hand the number of people entering a sensitive zone, with regards to a maximum in capacity acceptance, and on the other hand the number of people leaving the same zone. If the entering process exceeds the leaving one, this can lead to an overflow with regard to a certain threshold. An alarm could then be sent to the people in the zone, for instance by means of their smart phones, inviting them to avoid the zone, and to move to another safer place. For instance for the metro station, the smartphone could guide its owner to another entry/exit. The same holds for buildings like stadiums or museums involved in a special exhibition or match. In the field of tourism, a smart application could inform its user about the density of the queue of tourists as they want to visit some museum. A message is sent when a certain threshold is reached, meaning a too long waiting time before entering the museum. Technically the primitive tell(t(1)) can simulate somebody entering the zone or the queue, whereas the get(t(1)) primitive can simulate its exit of the same zone or queue. The overtaking of a definite threshold, i.e. 100, is obtained by a successful execution of an ask(t(100)) primitive.

The same reasoning can be used for a smart traffic management in cities. Most of them face an important incoming traffic in the morning, followed by a just as important outgoing one in

the evening. To tackle such traffic, a good strategy is to transform some road in a one way entry road in the morning and in a one way exit road in the evening. This requires to evaluate the density of traffic in both directions and to suggest to adapt the usage of a traffic line when a certain threshold is reached. Technically, the method to simulate the entering or leaving of one vehicle in the city can be established as in the previous example, with respectively the tell(v(1)) and get(v(1)) primitives. The overtaking of the threshold, for instance with a value of 500, can be tested with the ask(v(500)) primitive, in parallel to the tell and get primitives. Moreover, it is possible to distinguish the vehicles following their types, i.e. as they are cars, motorcycles, lorries or busses. For each type, the system could manage the traffic lights in order to drive them to some specific and more adapted roads. This could be typically the case for lorries transporting dangerous goods. In some circumstances, the intervention of the police could be requested.

In the field of public health, the capacity to measure the evolution of contamination is a key point for controling epidemics. In case of high-risk communities like elderly people, the fact to register the sick and healthy members and to measure their respective density when present in a common room permits to send warning messages to some of them, inviting them to leave the room. The message could consist in a colour code: red for danger of contamination, orange for a situation needing a careful attention, and green for an absence of danger. Technically the choice of the colour can be done with the following agent, working with the density of contaminated people in a room:

nask(t(10));tell(green) + ask(t(10));(nask(t(50));tell(orange) + ask(t(50));tell(red))

In this agent the success of the nask(t(10)) means that the number of sick people present is less than 10, and represents a weak possibility of contamination. Green is then the selected colour. A failure of the nask means a success for the ask(t(10)), meaning that there are at least 10 seak people present in the room. If the threshold for a definite contamination is fixed to 50, a success for nask(t(50)) means that the number of contaminated people has a value between 10 and 50. This situation means an average probability of contamination and requests a careful attention, which is expressed by the choice of an orange colour. On the other hand, a success of the ask(t(50)) means that the threshold is exceeded, and that the sanitary context is with high probability in red condition.

1.2 The thesis

The purpose of this thesis is to extend the Bach Language, a Linda-like language developed by the Coordinam Research Group of the University of Namur, with a notion of multiplicity, and to establish that the new obtained language has a net benefit able to meet the needs in development in certain domains as for instance service oriented computing, or recommendation systems, or measure of popularity.

Our thesis is that the introduction of the concept of density in the Bach language and the distribution of this density over a finite set of tuples produce new languages that are more expressive than Bach. More precisely, we claim that it is possible to enrich Bach with simple mechanisms, like the incorporation of a multiplicity of occurences of tokens manipulated by Linda on the one hand, and the distribution of this multiplicity on a list of tokens on the other hand. In spite of their simplicity, these operations provide new languages, that are strictly more expressive than Linda, but less than Gamma, a well-known coordination language inspired by a chemical metaphor. Nevertheless, the newly obtained languages preserve the advantage of an easier implementation than Gamma. The way we establish this increase of expressiveness provides us with a methodological framework that permits us to reason and establish the impact of a new language evolution.

1.3 Structure

The thesis is organized in four main parts. The first one presents the background material. It is itself composed of three chapters. After this introductory chapter, chapter 2 presents an overview of the family of coordination languages. Chapter 3 then introduces and studies the expressiveness of two coordination languages based on tokens, one, named BachT, being a variant of Linda and the other, named MRT, embodying the chemical metaphore.

The second part proposes and studies two languages. It is composed of four chapters. Chapter 4 defines our first language Dense Bach and provides several applications to evidence the interest of the language. The main feature of this language is to allow primitive operations to manipulate tokens in multiple instances. Chapter 5 defines our second language, named Dense Bach with Distributed Density. Its main feature is to equip primitives with means to distribute a number of instances on a list of tokens. After having identified the core mechanisms, different variants are there defined and applications are given to support their interest. The next two chapters study the expressiveness of the new languages. Chapter 6 studies the expressiveness of the Dense Bach language and relates it to the BachT and MRT languages. Chapter 7 studies

the expressiveness of Dense Bach with Distributed Density and relates it to Dense Bach and to MRT.

The third part focusses on programming aspects. It is composed of four chapters. The first two provide interpreters and command line simulators for four classes of languages. Chapter 8 deals with BachT and Dense Bach whereas chapter 9 refers to Distributed Dense Bach and MRT. A graphical simulator is then proposed in chapter 10 to provide the user with a support to experiment interactively with the executions of agents written in Dense Bach. Finally, through a modeling using Open Petri Nets, chapter 11 presents a framework to study the properties of programs written in Dense Bach, in particular to detect deadlocks.

The fourth part is composed of one chapter, which draws our conclusions and sketches future work.

1.4 Publications

Parts of this thesis have been the subject of three publications:

1. D. Darquennes, J-M. Jacquet and I. Linden. *On Density in Coordination Languages.* In C. Canal and M. Villari (eds), Proceedings of the ESOCC workshop on Advances in Service-Oriented and Cloud Computing, Series in Communications in Computer and Information Science, vol. 393, Springer-Verlag, pp. 189–203, 2013. [DJL13a].
   In this first paper, we propose an extension of Bach, aiming at promoting the notion of density associated with tokens and representing the simultaneous presence of a positive number of them. Based on De Boer and Palamidessi's notion of modular embedding, we establish that it strictly increases the expressiveness of Linda, while keeping the same implementation efficiency. We also study the hierarchy of the sublanguages induced by considering subsets of tuple primitives.

2. D. Darquennes, J-M. Jacquet and I. Linden. *On Distributed Density in Tuple-based Coordination Languages.* In J. Camara and J. Proenca (eds), Proceedings of the 13th International Workshop on Foundations of Coordination Languages and Self-Adaptive Systems, EPTCS, vol. 175, pp. 36–53, 2015. [DJL13b].
   In this second paper, we additionnally compare the Dense Bach language introduced in the previous paper with Gamma a language based on multiset rewriting. We establish

that, although it is less expressive, it benefits from a much more efficient implementation scheme. We study the hierarchy of the sublanguages induced by considering subsets of tuple primitives and prove that it follows that of the Linda family of languages.

3. D. Darquennes, J-M. Jacquet and I. Linden. *On the Introduction of Density in Tuple-Space Coordination Languages*. In Science of Computer Programming, vol. 115-116, pp. 149–176, 2016. [DJL14].

   In this third paper, we define a new extension of a Linda-like language in the aim of modeling the coordination of complex distributed systems. This language, called Bach with Distributed Density, manipulates finite sets of tuples and distributes a density among them. This new concept adds to the non-determinism inherent in the selection of matched tuples a non-determinism to the tell, ask and get primitives on the consideration of different tuples. Furthermore, like in the two previous articles, we establish that this new language strictly increases the expressiveness of the Dense Bach language previously introduced and, consequently, Linda-like languages.

Chapter 2

Coordination Languages and Models

2.1 Coordination as a natural evolution in Computer Science

Since its very beginning Computer Science has improved in many fields, like the reduction of the size of the hardware components and the increase in their speed of execution. Another natural axis of development has been the increasing degree of complexity. Indeed, the need to efficiently share limited and expensive computer resources and data between many users, or to increase the speed of computation, have contributed to bring to the foreground the development of more and more complex operating systems, and the parallelization of tasks on several processors. The emergence of the Internet and the connection of many dedicated networks have reinforced that trends. In turn, Internet applications – like web services – require to revisit traditional software architecture patterns, which results in a re-use and cooperation of distributed heterogeneous components.

Complexity arises when simple processes run in parallel to perform a joint task. It directly results from the communication those processes have to establish between them, in order to perform their tasks. This communication expresses a need of coordination between the different components acting concurrently. Gelernter and Carriero [GC92] postulate that the computation necessary for the tasks to be performed, and the communication required to coordinate the components, are *orthogonal*, meaning that they do not need to be incorporated in the same model. This introduces the concept of a programming model, constitued, in one part, by a coordination model, whose role is to glue together activities in an ensemble, and, in another part, by a computational model, whose role is to effectively perform the specifications of the

components. This is summed up in the following "equation":

Programming = Coordination + Computation

This vision facilitates, on the one hand, the re-use of a component, and on the other hand the re-use of patterns of coordination. Gelernter and Carriero proposed a communication model called the generative model ([Gel85]), and its implementation in Linda. In this generative model, communication of data is performed by generating new data objects, and by placing them in a shared dataspace from which the receiver can retrieve them.

Since the claim of Carriero and Gelernter, many coordination languages and models have been developed. To support their thesis Carriero and Gelernter themself proposed a language of coordination called Linda. But other languages have also been proposed, that are based on different approaches. To draw the broader setting of the thesis, we present subsequently the main properties of Linda and other languages, and classify them following [PA98].

The rest of the chapter is thus organized as follows. Section 2.2 concentrates on the Linda language as the first development in coordination. Section 2.3 provides a survey of the family of the coordination models and languages. Finally Section 2.4 concludes the chapter by a synthesis of its content, of its references and with a short supplement that extends the main presented languages.

2.2 Linda as the first coordination language

Linda is the first language that has been classified as a coordination language, i.e. being an embodiment of a coordination model, expressing the interaction of autonomous agents within some environment and other agents that are included in. Despite the fact that it provides only four primitives, Linda is able to deal with the complexity of coordination, to build parallel applications, to design distributed computing platforms, and to program agent based systems.

To start, this section briefly presents the notion of tuple. Then a description of the four basic primitives that constitute Linda is provided. Finally, the section concludes by highlighting

the characteristics of the generative model.

2.2.1 The tuples

At the base of Linda is the shared dataspace model, also called the coordination medium. It contains tuples, that are ordered sequences of data. Those tuples constitute messages, around which the communication and cooperation is organized. In Linda the communication is performed asynchronously. Messages produced by a sender are collected in the dataspace, and are made equally available to any other consulting agent. The latter can then access it by reading or removing it from the dataspace, by means of a template specifying the kind of messages it is interested in. One available tuple matching the template is retrieved or read from the dataspace.

2.2.2 The primitives

Linda provides a set of four primitives. Three of them (*out*, *in*, *rd*) are grouped in a category for managing the messages: *out*, *rd* and *in* are respectively responsible for inserting tuples on the dataspace, for reading them on the dataspace, and for retrieving them out of the dataspace. In case the required tuple is not available on the dataspace, the *in* and *rd* operations adopt a blocking behaviour, until it is present. It can be argued that the read operation can be considered as semantically redundant as it can be modelled by an input *in(a)* immediately followed by an output *out(a)* of the same datum *a*. Although this first view looks intuitive, we shall nevertheless see that it needs to be revisited when we shall study the expressiveness of languages.

The last primitive (*eval*) is part of the second category and is responsible for agent and process generation. More precisely, with such a primitive, tuples that are basically passive entities, can now acquire an active behaviour. These tuples contain one or more functions as parameters. When *eval* is invoked, an active tuple is created and its parameters functions are evaluated in parallel with each other and with the calling process. When the computation of the function is finished, the results are inserted in the tuple, that then becomes a passive one, which is placed into the tuple space.

Two characteristics of the tuple access are worth being emphasized. Firstly, the *out*, *in* and *read* primitives access the tuples in an associative way. This means that the search relies on the use of a template, that expresses the kind of tuples an agent is interested in. There is thus no

need to specify a memory address. Secondly, any emitted message placed in the dataspace has an independant existence since it is retrieved asynchronously by a receiver. Until that moment, the message is in fact equally accessible to all agents. This implies that in case of the simultaneous presence of multiple identical tuples on the dataspace, any consultation or withdrawing of one instance of this kind of message, is done in a random way.

2.2.3 The generative communication model

In place of the simple approach of considering some low level primitives like *send* and *receive* to transmit messages over channels, communication in Linda is based on a shared space of data, accessible to any agent accessing it. In this sense, Linda is said to be generative, as the agents communicate by generating data on the shared space. As a result, in Linda every agent is potentially able to communicate with all the others, at the contrary of the message-passing paradigm, that is a private act between some participating agents, having to share some channels. As well as third-party agents, the dataspace itself is also able to manipulate the messages present in itself. The dataspace can have a complex structure, for instance in the form of multiple nested dataspaces, and, in most cases, is complemented by event-based mechanisms.

2.3 A survey of the family of coordination models and languages

2.3.1 General concerns

Many coordination models and languages have been developed after Linda. A great number of conferences (e.g.,[Sha92, GFM04, JP05, CW006, CA10, MR11a, Sir12, NJ13, KP14, HV15, LP16, JM17]), conference tracks (e.g.,[Pan02, CDH00, CW111, SM113, WC115]), workshops (e.g.,[BJ03a, BJ03b, BJP04, CV06, MS10, MR11b, KR12, CV13, CP15]) and journal special issues (e.g., [BJKP06, BJK06, JLD16]) attest of the enormous amount of research in this area. In [PA98], Arbab and Papadopoulos have proposed to classify coordination languages into two main categories: the first one works through an exchange of data through a shared memory and is called the "data-driven" category. The second one works through an exchange of messages between the entities and is called the "control-driven" one, or also the process- or task-oriented one.

Data-driven coordination models and languages are characterized by the fact that the state of the computation at any moment in time is defined in terms of both the values of the data

being received or sent and the actual state of the coordinated components. In other words it is the availability of the data produced and consumed by the data-driven coordinated computation that determines its progress. Regarding the equation proposed by Carriero and Gelernter [CG89], the separation between computation and coordination is done at the level of functionality. Those can be of two types: either purely coordinational, with the use of coordination primitives, or purely computational, both types being possibly used by the same process. Two of the most famous languages of this family are Linda, already presented in the previous section, and Gamma that will be presented in section 2.3.3.

Control-driven coordination models and languages are characterized by the fact that coordination and computation are achieved by distinct agents. In opposition with the data-driven family, the value of the data being manipulated by the processes are of no more importance in the definition of the state of the computation, that is only defined in terms of the coordinated patterns. Manifold and its successor Reo are two representatives of this family of coordination languages, and will be presented in section 2.3.2

In a more constructive perspective, and following [BCGZ01], the design of a coordination language has to meet three kinds of issues: the coordination of the entities, the determination of the media for coordination, and the protocols and rules used for coordination.

The first one concerns the necessity to coordinate entities, also called agents or processes, that are usually actively computing entities programmed in many different languages. Despite this variety, the coordination of the agents should not require their re-programming, but in place should wrap them all in an ensemble. The second issue to be encountered is to determine the media for coordination. In place of the simple approach of considering some low level primitives like *send* and *receive* to transmit messages over channels, the communication proposed by Linda is based on a shared space of data, accessible to any agent accessing it. The third and last issue concerns the protocols and rules that are used for coordination. Those coordination rules regulate the relationship between the coordinable agents and the coordination media, and may be expressed either in an operational way, or in a more abstract and declarative way. Linda proposes a minimal set of primitives, implemented as library routines invoqued by some host programming languages.

Those three issues help to consider three key concepts for the definition of a coordination model. They are at the basis of the three following questions :

1. What are the entities to be coordinated ?

2. What is the considered coordination medium ?

3. What are the corresponding coordination rules ?

Those three concepts - respectivelly the coordinables, the coordination medium and the coordination rules - can be represented in a formal framework, that can be instantiated to a variety of coordination models and languages for agents. In order to capture the main aspects of many other models, the dataspace-based model of Linda will be extended in the three following directions:

1. the more advanced coordination primitives exploitable by the coordinables,

2. the reshaping of the coordination medium, and,

3. the programming of the coordination rules.

With respect to the coordinables, three kinds of new primitives can be added to the four basic ones of Linda. The first type is the one that permits the computation of transaction operations involving more than one datum. They can be transactions, that are implemented for instance in JavaSpaces ([FHA99]), or multiset primitives, that atomically produce and /or consume multiset of data, as the *multiwrite* primitive in TSPaces ([Wyc98]), or reactions resulting from the inspiration of the chemical metaphor, considering the items in the dataspace as molecules moving freely in a chemical solution. Gamma is an example of language resulting from that natural science inspiration.

The second type of new primitives is the one that requires for the primitive to be executed, a global vision of the state of the shared repository. Examples of such primitives are given by the *count* operation that, in TSpaces, returns the actual number of data inside the repository satisfying a given condition, or by a test-for-absence of data, that verifies the non-availability of data of a certain kind. In order to bypass the limitation of the blocking bevaviour of its *in* and *read* primitives, Linda itself has been extended with their nonblocking version, respectivelly *inp* and *readp*, both requiring a global view of the datapsace.

Finally the third type results from a combination of the two previous ones, providing global operations able to perform tests on the global state of the shared repository. The *collect* primitive, which removes all the data satisfying a specified pattern, is an example of such an operation.

As for the coordinables, a re-shaping of the coordination medium in the form of a collection of either named independant spaces as in KLAIM ([RDP98]) and TuCSoN ([OD01]), or structured nested spaces as in Bauhaus Linda ([CGZ95]), or merged spaces as in LIME [GPR99], is the

second possible way for extensions of coordination models or languages. The introduction of multiple spaces has the advantage of providing modularity by restricting the visibility to data present in a particular dataspace. Allowing a network-aware style of programming by allocating spaces to a particular node of the net, is a second advantage.

Finally, allowing a dynamic modification of the coordination rules to obtain a programmable coordination medium is a third way to introduce new extensions in the coordination models. Such an introduction of programmability has always for objective to enable some form of the control of the access to the shared repository, and to allow for reactions in case of violation of contracts. TuCSoN is again an example of a language providing this capability.

The next sections are dedicated to the presentation of some principal languages, among the wide variety of the coordination language family. In a first step, we present the languages Manifold and its successor Reo, as typical examples of the process-oriented models. Then Gamma and TuCSoN will be briefly explored, as examples of languages based on the tuple-space model, Gamma illustrating the chemical reaction model and TuCSoN, the enhancement of coordination rules by programming the dataspace. KLAIM and LIME are presented as examples of formalisms aiming at tackling mobility. Thereafter, we consider the introduction of quantitative information in the tuple spaces, and extension of Linda with priorities or probabilities.

2.3.2 Manifold and its successor Reo

2.3.2.1 Manifold

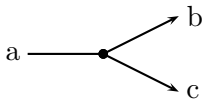
Channels of communication are used as an alternative to the shared dataspace of Linda in some coordination models. These models rest on the idea of an *Idealized Worker* and an *Idealized Manager* (IWIM) ([Arb96]). The language Manifold is an illustration of this approach and consequently is an example of a control-driven language. Making a clear and complete separation between computation and coordination at the level of modules or processes, it provides on the one hand "worker" entities, that are responsible for the computation, and on the other hand "manager" entities, that are responsible for communication and cooperation. The manager is the only one to know all the workers, to dynamically connect output ports of workers to their input ports and to modify these connections. This exclusive property makes possible a complete anonymity at the worker level, which can then perform computations without taking care of other components.

Manifold uses input and output ports to organize communications between different processes and their environment. Asynchronous channels called streams create the links between output and input ports of worker entities. Coordinator processes are able to dynamically modify these streams between the workers. The information exchange is taken in charge by an event mechanism. The occurrence of some relevant events triggers the concerned coordination processes, placing them in a state to perform some actions.

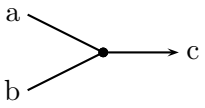
2.3.2.2 Reo

Still in the world of channel-based coordination models, Reo ([Arb04]) is the successor of Manifold. It is used as a language for the coordination of concurrent processes, or to orchestrate component instances, like fragments of sequential code, threads, objects, agents, in a component-based system. Most of the time, the interfaces of these components do not fit perfectly with each other. For this reason there is a need for a so called specific "glue code" to permit a correct matching. The idea of Reo is to construct this glue code in a compositional way, using atomic primitives. Channels constitute those primitives. Every channel is a point-to-point medium of communication, caracterised by its own unique identity and by two ends, that can be either a source or a sink. A source channel end accepts data into its channel, and a sink channel end dispenses them out of it. Moreover to be communication paths, channels also impose relational constraints, like synchronisation, buffering, mutual exclusion, or even lost on the data flowing through their ends. From [Pro11], we present in Figure 2.1 some of the most commonly used Reo primitives.

In addition Reo defines the operations that permit to plug together the ends of the channels, in order to form complex connectors. A connector is a set of channel ends and their connecting channels organized in a graph of nodes and edges. A node is a logical place where channel ends coincide. Due to the existence of two types for the channels, a node can consist of a sink end, or a source end, or both a sink and a source ends. Following Proença ([Pro11]), nodes with a single end are called *boundary nodes*, and are represented by $\circ$. Nodes with both a sink and a source ends are called *mixed nodes*, and are represented by $\bullet$. Within a connector, data will flow from primitive to primitive through nodes, without being buffered by them. This means that both ends in a node are synchronised and have the same dataflow.



Replicator. This channel replicates the data (without corrupting them) in a to all of its ends (namely b and c) or nowhere. Here represented as a 2-replicator, it can be extended to an n-replicator, with n sink ends.

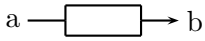


Merger. The function of this channel is to copy data synchronously from a or b to c, excluding a simultaneous incoming from a and b. If this situation occurs, the origin of the data flow is then chosen in a non-deterministic way.

Sync. This primitive represents a synchronous channel. It accepts data at its source end if it can simultaneously transfer them at its sink end.

LossySync. This primitive differs from the synchronous channel by the fact that it always accepts any data coming into its source channel. Nevertheless the transfer of the data to the sink end is done only if they can be dispensed through.

SyncDrain. The function of this channel is to synchronise both source ends a and b. Data will flow at one end if and only if they also flow on the other end.



FIFO$_1$. This primitive represents a one-place buffer channel, caracterised by two possible states : empty or full. If empty, the buffer can receive a data item from a, changing its state to full. Then no more data can be received anymore, but the channel can transfer the previously received data to its end b, resseting the state back to empty.

Figure 2.1: The most commonly used Reo primitives.

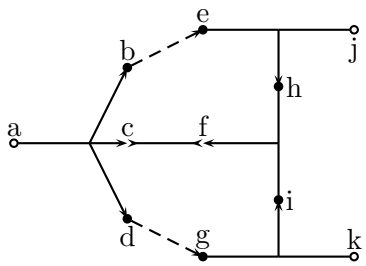


Figure 2.2: Representation of an exclusive rooter.

In the following we present some simple examples of connectors built thanks to the previously introduced primitives. The first example (from [Pro11]) in Figure 2.2 describes an exclusive router. It is made of three replicator primitives represented by *a*-*b*-*c*-*d*, *e*-*h*-*j* and *g*-*i*-*k*, of two LossySync channels represented by *b*-*e* and *d*-*g*, of one Merger *h*-*i*-*f* and one SyncDrain *c*-*f*. Every data A entering the router in *a* is first replicated on *b*, *c* and *d*. If the data crosses through the LossySync channel *b*-*e*, the synchronisation with the SyncDrain channel *c*-*f* permits to transfer the data A to *j*. A symmetric behaviour is also possible with the LossySync channel *d*-*g*, leading to produce the data A at *k*. In case data A crosses simultaneously through both LossySync channels *b*-*e* and *d*-*g*, then the Merger *h*-*i*-*f* makes a non-deterministic choice. In every case, data can never flow from *a* to both *j* and *k*.

The second example uses the previous exclusive router and represents a connector that eliminates one element over two in a string of characters. Figure 2.3 represents the schema of this connector, with the exclusive router represented by a circle in place *b*. When a string of characters "AB" presents itself at the boundary node *a*, the first letter A is sent either to the node *c*, or to the node *d*, by the exclusive router located in *b*. In case the letter A is sent to *d* then it is stored in the FIFO_1 buffer *d*-*f* and is blocked this way. The second letter B can then only be sent through Sync channels *b*-*c* and *c*-*e*. At node *e* it waits for the SyncDrain channel *e*-*f*. The only possiblity for continuing is then to wait for the buffer to liberate the letter A. The letter B can then continue after the action of the SyncDrain channel *e*-*f*. If the letter A is sent to *c*, the process will work the same way, but now producing the letter A on channel *e*-*g*.

2.3.3 Gamma

The chemical metaphor has been used as a basis of inspiration to develop coordination models. Gamma ([BL93, BM96]) embodies this metaphor. It considers the items in the dataspace as

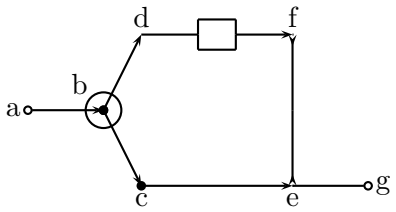


Figure 2.3: Connector eliminating one element over two.

$$\text{ChemicalReaction(M)} = \Gamma((\text{R,A}))(\text{M}) \text{ where}$$
$$\text{R(x,y,z)} = (\text{ x} = \text{y} = H_2, \text{ z} = O_2 \text{ })$$
$$\text{A(x,y,z)} = \{\ H_2O,\ H_2O\ \}$$

Figure 2.4: The chemical reaction $2H_2 + O_2 \longrightarrow 2H_2O$ in Gamma.

molecules that freely move in a chemical solution. Reactions between those molecules can happen, when they come in contact, and also satisfy certain constraints. Gamma programming style sees atomic values regrouped in one single bag, and computes the results of their individual interactions, following the locality principle. This stipulates the independance of reactions between values to produce new ones. As a consequence, Gamma programs are caracterized by a minimum of explicit control.

More explicitely, a Gamma program is a pair (R,A), with R a *reaction condition*, namely a boolean function on multisets of data, and A a *rewriting action*, namely a function from multisets to multisets of data. Following this language, any group of molecules satisfying the reaction condition can be rewritten according to the corresponding rewriting action.

An example of program written in Gamma is the one describing the chemical reaction of production of water molecules, using molecules of hydrogen and oxygen. The full program is presented in Figure 2.4. In this program, M is a multiset of data that are molecules composed of a certain number of atoms. The reaction R needs a certain number of molecules, each of them being allocated to variables.

The reaction condition R expresses that two molecules of hydrogen and one molecule of oxygen are necessary to activate a chemical reaction. R has then three variables x, y and z, where x and y receive the same molecule of hydrogen H_2 and z receives the molecule of oxygen

O_2. Under these conditions, the rewriting action A transforms the two molecules of hydrogen and the molecule of oxygen in two molecules of water.

2.3.4 TuCSoN

TuCSoN ([OZ]) is based on the same set of primitives as Linda to act on a tuple space, either for putting (*out*) a tuple, or for reading it with or without suspension (*rd*, *rdp*), or for retrieving it, still with or without suspension (*in*, *inp*). TuCSoN extends Linda in two ways: firstly by introducing multiple tuple spaces, referred to as tuple centers, and secondly by providing the possibility to program them, in order to define locally the way in which agents interact with them. The consequences of these extensions are respectively the parameterization of classical primitives to the particular tuple center which is being accessed, and the handling of the queries by the medium, in addition to its access by the agents. In TuCSoN, coordination is expressed by the specification of the reactions of the tuple center to some events. ReSpecT (Reaction Specification Tuples) ([OD01]) is a logic-based language that specifies these reactions. Two primitives are used to program a tuple center behaviour: they are *set_spec* and *get_spec*, respectively responsible for adding and removing logic tuples. The syntax of these tuples is *reaction(Op,(Body))*, expressing the association of a communication operation *Op* to the *Body* reaction body. Every performed operation *Op* sees all its corresponding reactions triggered, and their reaction bodies executed. The reaction *Op* is defined on the one hand for ReSpecT primitives like *out_r*, *rd_r*, *in_r* and *no_r*, where *out_r* works as a conventional *out*, while *rd_r* and *in_r* have the same non-blocking effect as *rdp* and *inp*, and finally *no_r* acts as a test for absence. On the other hand, it is also defined for operations over tuple spaces performed inside reactions. The reaction *Body* is a sequence of *ReSpecT* predicates with a failure/success transactional semantics. Among those predicates are internal primitives like *out_r*, *rd_r*, *in_r* and *no_r*, as well as constructs such as also *pre*, *post* and '*X is expression*'. Both predicates *pre* and *post* succeed in their respective *pre* and *post* phases of any operation. The *is* predicate is an assignment.

A first example of the use of TuCSoN (from [Tuc04]) consists in writing reaction rules for keeping track of the number of tuples inside the tuple centre. This is achieved by maintaining on the data space a tuple n_tuples(N), with N indicating the numbers of tuples present in the tuple centre. Every tuple added or removed from the space must trigger an update of N, with respect to the specifications of the following rules:

reaction(out(X), (in_r(n_tuples(N)), N1 is N+1, out_r(n_tuples(N1)))).
reaction(in(X), (in_r(n_tuples(N)), N1 is N-1, out_r(n_tuples(N1)))).
reaction(inp(X), (in_r(n_tuples(N)), N1 is N-1, out_r(n_tuples(N1)))).

The second example, also taken from [Tuc04] concerns a coordination artifact needed in concurrent systems, called the synchronisation barrier. In this example, N agents have first to synchronise before to continue their individual activities. Basically every agent executing its task and reaching the synchronisation point has to wait for all the other agents to reach the same point before to continue. A solution consists for an agent to insert a ready tuple when finishing its activity, and then to wait for a ready_all tuple before continuing. This is expressed by the following protocol:

out(ready).rd(ready_all).

Then the tuple centre acting as a synchronisation barrier obeys the two following specifications:

```
reaction (out(ready),
        (in_r(ready),
        in_r(ready_agents(N)),
        N1 is N+1,
        out_r(ready_agents(N1)))).

reaction (out_r(ready_agents(N)),
        (rd_r(barrier_size(N)),
        in_r(ready_agents(N)),
        out_r(ready_agents(0)),
        out_r(ready_all))).
```

The first rule clarifies the reaction of the tuple centre when an agent inserts a ready tuple as he finishes his activity. The tuple centre simply increases the number N of ready agents, and then invokes the second rule, with this new value for N. The second rule first questions the tuple barrier_size, that contains the current number of agents needed to be synchronised. Then it takes a tuple ready_agent that contains the dynamic counts of agents already synchronised. The last two steps reset to 0 the number of ready_agents, and publishes on the tuple centre the tuple ready_all.

TuCSoN has been designed for the development of efficient applications for the access to heterogeneous and dynamic information distributed over Internet. The reader interested in more details is referred to [OD01].

2.3.5 Klaim

Mobilty can be understood both from the logical and physical points of view. The first aspect reflects the migration of code or data through networks. The second is concerned with applications distributed on mobile devices. In the following two sections we consider two coordination languages that respectively encounter those two understandings of mobility: KLAIM and LIME. KLAIM is designed to handle the physical distribution of processes through a wired network. LIME is designed to manage the coordination of communicating mobile applications over both wired and ad hoc networks.

KLAIM is the abreviation for the Kernel Language for Agents Interaction and Mobility ([RDP98]). It consists of a Linda variant with multiple tuple spaces. KLAIM is able to manage the physical distribution of processes over networks, as well as to control changes in their configuration. To that end, KLAIM considers a finite set of physical localities and associates nodes with them. These nodes are composed of three elements: a *site*, a *process* and an *allocation environment*. Inside this triple, every site component is associated with a tuple space. As regards the allocation environment, its function is to associate some (logical) *localities* with (physical) *sites*. KLAIM extends the classical Linda primitives *out*, *in*, *read* and *eval* as they not only can be invoked locally on the tuple space located on the same site, but also on a tuple space of a remote site specified by providing the locality of a node to the primitive invocation. KLAIM provides also a specific primitive *newloc(u)* for binding a fresh site with a variable *u*. The implementation of the mobile code paradigms is obtained by a combination of *newloc* with a remote invocation of primitives as *eval*.

2.3.6 Lime

In a mobile environment, the shared tuple space properties of persistency, global accessibility and static creation are no longer maintained. The model of LIME ([GPR99]) adapts the basic philosophy of Linda to mobility, by transforming the shared tuple space into a *transient shared tuple space* (TSTS), and by associating an *interface tuple space* (ITS) with each process. The ITS contains the information that the process is willing to share with others; it is accessible by means of the classical Linda primitives *out*, *in* and *read*. It is the combination of the ITS of

the currently co-located mobile processes that constitutes the TSTS seen by all those processes. During the computation, the content of the TSTS is dynamically modified, as the co-located agents become different. The management of this dynamic recomposition of the TSTS due to the change of connectivity is taken in charge by the Lime middelware and is completely hidden to the processes.

LIME also proposes, for some extended primitives, to specify the interface tuple space with which they want to interact. As a result a process can write on the interface of another distant process, by putting on its own interface tuple space the tuple to be sent, tagged with the destination process identifier. Under the condition that the target interface is reachable, the LIME middleware migrates the tuple. Otherwise the tuple stays on the interface of the sending processes until the target becomes reachable. For the *in* and *rd* primitives, it is also possible to restrict their search to the actual contents of some specified interface tuple space, or to tuples with, as destination, a specific interface tuple space, both criteria being simultaneoulsy applicable.

2.3.7 Linda with priorities or probabilities

Several pieces of work have investigated the introduction of quantitative information in the tuple spaces. In this section, we present the most relevant ones for our work and compare their method with the one we follow.

Some authors ([BGLZ04, BGLG05]) have investigated the impact of the introduction of quantitative information in the tuple space coordination model in order to quantify the relevance or importance of each tuple. They do this at the syntactical level by decorating each tuple with an extra field and at the semantical level by providing two possible interpretations for this information. On the one hand, the field quantifies a weight indicating how frequently each tuple should be retrieved, while on the other hand, it expresses a priority level for the tuples.

An example (from [BGLG05]) concerns a master-worker application, with a master producing jobs executed by workers. The master produces job requests and stores them as tuples in the tuple space. The workers then question the same space, in order to retrieve the description of jobs to be executed. However due to non-determinism, it is possible to observe some jobs that will never be selected. Moreover this way to work also ignores a natural priority for execution that is requested by some specific jobs, with regards of every all other jobs. These two scenarios find an easy solution within the possibility of decorating tuples with an urgency level.

Other authors ([VC09, VC10]) have proposed a stochastic extension of the Linda framework. This model associates tuples with an activity/pertinency value, similar to the notion of chemical

concentration. It measures to which extent the tuple can influence the coordination state. The syntax of this tuple space is modeled by means of a calculus, with an operational semantics given as an hybrid CTMC/DTMC model.

An example developed in [VC09] proposes a model of chemical tuple space. In this model an integer value called *concentration* is attached to each tuple, measuring its pertinency/activity, i.e. the frequency with which it can influence the system coordination. The *out* primitive serves to inject a tuple with any initial concentration. In the presence of the same tuple in the space, both tuples join and their concentration are summed. The *in* primitive is either able to entirely retrieve a tuple, or to decrease the concentration of an existing tuple. The first situation occurs when there is no specified concentration with the primitive, and the other when there is well a specified fixed concentration. The *read* primitive *rd* acts as the *in* primitive, in order to discover the concentration associated with a tuple.

For instance the following instruction expresses the retrieval of 10 tuples among three different ones, a or b or c. Those tuples are present on a store σ , respectively with a concentration of $\langle 20 \rangle$, $\langle 10 \rangle$ and $\langle 5 \rangle$. The primitive questions the store σ for every name, as the latter is represented by variable X, and with a concentration of $\langle 10 \rangle$.

$$[\![t(a)\langle 20 \rangle \,|\, t(b)\langle 10 \rangle \,|\, t(c)\langle 5 \rangle]\!]_\sigma \,|\, in(\sigma, t(X)\langle 10 \rangle).0$$

As the concentration of c is only of 5, this latter tuple cannot match the request. For a and b, the decrease will be of 10 units. As a is present two times more than b, the probability for a to be concerned is two times higher than for b, i.e. in this case respectively of 2/3 and 1/3.

The aim to entirely remove a tuple is reached by introducing inside the primitive *in* a variable v for the concentration. For instance

$$[\![t(a)\langle 20 \rangle \,|\, t(b)\langle 5 \rangle]\!]_\sigma \,|\, in(\sigma, t(X)\langle v \rangle).0$$

means that it is equally probable to totally remove *t(a)* unifying v with 20 and X with a, or to remove *t(b)* unifying v with 5 and X with b.

S. Mariani and A. Omicini ([MO13]) have introduced the concept of uniform primitives. Their analysis concerns the data-retrieval primitives, i.e *in* and *read* primitives - also called *getter* primitives, that are shared by all tuple-based coordination models. The uniform primitives are a response to the inherent *don't know non-deterministic* character of the *getter* primitives. Indeed for the latter, which tuple among the matching ones is actually retrieved can be neither specified, nor predicted. In other words, they only ensure a *point-wise* property for a tuple retrieval, both

in space and time. In terms of spatial context, a single getter operation can return a matching tuple independently of the other tuples currently in the same space. For the time context, a sequence of getter operations presents no meaningful properties.

Instead, the uniform coordination primitives *uin* and *urd* are a specialisation of the getter primitives of Linda, but still compliant with their standard semantics. In place of the *don't know non-determinism*, they feature a *probabilistic non-determinism* with uniform distribution, ensuring global system properties in place of point-wise ones. For a single getter operation, they return matching tuples based on the overall state of the tuple space, and for a sequence of getter operations, they tend to a uniform distribution over time.

A. Di Pierro and al ([PHW05b]) have considered how probabilities or quantities may be added to a Linda-like language. Their analysis follows the classification of the coordination languages into the two categories presented in section 2.3.1 : on the one hand the "data-driven" category and on the other hand the "control-driven" category. For the first category, positive natural numbers called *priorities*, or positive real numbers called *weights* are added as an attribute of tuples. Based on this attribute, a retrieved tuple is the one with the higher priority for the priority approach, or is the one selected with a probability proportional to its weight for the probability approach.

In a dual way, the control-driven approach proceeds by adding priorities or probabilities to the operators, in particular to the parallel operator. This approach requires to define the notion of "active state", which identifies the processes that are able to make a transition, i.e. that are not blocked awaiting for a tuple to become available. Replacing the parallel composition $P|P$ by a prioritised parallel operator, $p_1 : P_1|p_2 : P_2$, with p_1 and p_2 positive natural numbers, a scheduler will non-deterministically select the state with higher priority among the active ones.

2.4 Conclusion

This chapter has sketched various coordination languages. The need for these languages stems from the growing complexity of Computer Science, mainly characterized by a concurrent collaboration of communicating processes in order to perform a joint task. Gelernter and Carriero [GC92] have postulated a clear separation between the computation activities of each components, and the coordination dialog between them in order to glue their activities in an ensemble. Since this original proposal many languages of coordination have been developped.

We have presented a non exhaustive review of the principal ones. Based on the work "Coordination of Internet Agents" [BCGZ01], we have listed the most representative languages among

the family of coordination. Despite their great number, and following [PA98] the languages can be regrouped in two main categories : the "data-driven" one, and the "control-driven" one. The data-driven category makes use of a shared memory to exchange data between the processes, while the control-driven category exchanges messages directly between the entities to be coordinated.

Linda [Gel85] is the first coordination language and belongs to the first category. It is caracterised by a basic set of four primitives, mainly responsible for managing the exchanges with the shared data-space. Gamma [BL93, BM96] is another language of this family, inspired by a chemical metaphor. It is based on a data structure which is a multiset, and is mainly caracterised by a high level nature, with a very abstract description of programming and a natural construction of parallel programs. TuCSoN [OZ, OD01] extends Linda by introducing multiple tuple spaces, and by giving the possibility to program them, in order to define locally the way in which agents interacts with them. Klaim [RDP98] and Lime [GPR99] are concerned with mobility. The first one handles the physical distribution of processes through a wired network. The second one is more concerned with the management of the coordination of communicating mobile applications over both wired and ad hoc networks.

Reo [Arb96, Arb04, Pro11] is certainly the most representative language in the control-driven category. It is a channel-based coordination language, where channels are point-to-point medium of communication. Complex connectors are built in a compositional way out of these simple channels.

The last subsection of our survey has explored some extensions of Linda aiming at introducing the concepts of priorities or probabilities. Bravetti and all [BGLZ04, BGLG05] add quantitative information in the tuple space in order to quantify the importance of each tuple. Viroli and Casadei [VC09, VC10] have proposed a stochastic extension of Linda. Both their approach need to add an extra field to the tuple structure. Mariani and Omicini [MO13] have introduced the concept of uniform primitives, essentially for the *in* and *read* primitives. In response to the non-deterministic behaviour of the latter, they propose respectively two primitives *uin* and *urd* featuring a probabilistic non-determinism. Di Pierro et al [PHW05a] have also investigated how to introduce priority or probability to languages from both families: the data-driven and the control-driven. For the first category, positive natural numbers or positive real numbers are added to the tuples; they are respectively called priorities and weights. The choice of the selected tuple is based on the highest priority, or is probabilistically selected based on the weight. For the control-driven approach, the same pieces of information are added to the parallel operator. The most active state is then selected based on these pieces of information.

As the reader will appreciate in the next chapters, we shall introduce a notion of multiplicity for Linda-like languages, similar to the above notions of priorities or probabilities. However, several differences will be pointed out, one of which being the fact that we do not add an extra attribute to the tuples.

Other developments of coordination languages have been proposed, for instance in the domain of the introduction of temporal modalities in coordination languages. The articles [LJBB04, LJ04, LJ07, Lin07, JBB00, JL07, JL09, ILJ11] are examples of this trend, mainly based on the so-called two-phase time approach.

The next chapter concentrates on the scientific material necessary to understand the thesis: it details a version of Linda and Gamma and introduces our methodology to compare the expressiveness of languages while applying it to versions of Linda and Gamma.

Chapter 3

Variants of Linda and Gamma

In order to subsequently introduce the languages we shall study in the thesis (both from the design and the semantical points of view), this chapter introduces two simplified coordination languages: a Linda-like one and a Gamma-like one. Both use tokens instead of tuples in the aim of highlighting the core features of our work. We also introduce a methodology to compare the expressiveness of languages and apply it on the coordination languages just introduced.

3.1 BachT and MRT: two coordination languages

We first consider a simplified version of a dialect of Linda, developed at the University of Namur, named Bach (see [JL07]). This language is based on four primitives for accessing a tuple space. The *tell(t)* primitive puts an occurrence of the tuple t on the tuple space. The *ask(t)* primitive checks the presence of the tuple t on the tuple space while the *nask(t)* primitive checks its absence. Finally, the *get(t)* primitive removes an occurrence of the tuple t on the tuple space. It is worth noting that the *tell* primitive always succeeds whereas the last three primitives suspend as long as the presence/absence of the tuple t is not met. Moreover, the tuple space is seen as a multiset of tuples, which naturally leaves room for multiple occurrences.

The simplified version we shall use is based on tokens instead of more structured tuples to strenghten our focus on the coordination patterns. As a result, to further stress this point, we shall use *store* for the corresponding tuple space, as it is actually composed of tokens. The resulting language is subsequently denoted as BachT. It is formally defined as follows.

Definition 1. Let *Stoken* be an enumerable set, the elements of which are subsequently called tokens and are typically represented by the letters t and u. Let define the set of stores *Sstore* as the set of finite multisets with elements from *Stoken*.

Definition 2. Define the set $\mathcal{T}$ of the token-based primitives as the set of primitives T_b generated by the following grammar:

$$T_b \quad ::= \quad tell(t) \mid ask(t) \mid get(t) \mid nask(t)$$

where t represents a token.

The second language is a Gamma-like one. As noted in chapter 2, this approach is inspired by a chemical reaction metaphor, where the perception of the data is more global.

Gamma considers communication primitives as the rewriting of pre-condition multi-sets into post-condition multi-sets. Intuitively, the operational effect of a multi-set rewriting (*pre*,*post*) consists in inserting all the positive post-conditions, and in deleting all the negative post-conditions from the current space σ, provided that σ contains all positive pre-conditions and does not meet any of the negative pre-conditions. Formally, these rewritings are specified as follows.

Definition 3. Define the set of multi-set rewriting primitives $\mathcal{T}_{MR}$ as the set of primitives T_{MR} generated by the following grammar:

$$\begin{aligned} T_{MR} \quad &::= \quad (\{M\}, \{M\}) \\ M \quad &::= \quad \lambda \mid +t \mid -t \mid M, M \end{aligned}$$

where λ indicates an empty multi-set and where t denotes a token.

It is worth observing that not all pairs of preconditions and postconditions correspond to reasonable computations. Indeed, as stated above, it is possible to require in a precondition that the same token is present and absent or to require in the postcondition the removal of a token which has not been tested for presence in the precondition. We subsequently define such reasonable pairs of pre- and post-conditions as respectively consistent and valid. To that end, we first introduce some notations.

Definition 4. Given a multi-set rewriting pair $(Pre, Post)$, denote by Pre^+ the multi-set $\{t \mid +t \in Pre\}$ of tokens positively appearing in the precondition and by Pre^- the multi-set $\{t| -t \in Pre\}$ negatively appearing in it. Similarly, we shall denote by $Post^+$ and $Post^-$ the multiset of tokens appearing positively and negatively in the postcondition.

A multi-set rewriting pair (*Pre*,*Post*) is said to be *consistent* if $Pre^+ \cap Pre^- = \varnothing$. It is said to be *valid* if $Post^- \subseteq Pre^+$.

A consequence of consistency and validity is that four basic pairs of pre- and post-conditions can be put forward: $(\{+t\}, \{\})$, $(\{-t\}, \{\})$, $(\{\}, \{+t\})$, $(\{+t\}, \{-t\})$. They correspond respectively to the $ask(t)$, $nask(t)$, $tell(t)$ and $get(t)$ of the BachT language.

We are now in a position to define the two languages BachT and MRT. To grasp their complete version, we shall define them by considering as complex agents the statements obtained by combining them by the non-deterministic choice operator "+" (used among others in CCS), the parallel operator, denoted by the " || " symbol, and the sequential operator, denoted by the " ; " symbol. The formal definitions are as follows.

Definition 5. Define the BachT language as the extended set of agents A generated by the following grammar:

$$A \quad ::= \quad T_b \mid A \,;\, A \mid A \,||\, A \mid A \;+\; A$$

where T_b represents a token-based primitive, and where " ; ", " || " and " + " denote the sequential, parallel and choice compositional operators.

Similarly we define the multi-set rewriting language MRT by taking consistent and valid multi-set-based primitives T_{MR}:

Definition 6. Define the MRT language as the extended set of agents generated by the following grammar:

$$A \quad ::= \quad T_{MR} \mid A \,;\, A \mid A \,||\, A \mid A \;+\; A$$

where T_{MR} represents a valid and consistent primitive as defined in definitions 3 and 4, and where " ; ", " || " and " + " denote the sequential, parallel and choice compositional operators.

Subsequently, we shall consider sublanguages formed similarly but by considering only subsets of primitives. In that case, if $\mathcal{H}$ denotes such a subset, then we shall write the induced sublanguages as $\mathcal{L}_B(\mathcal{H})$ and $\mathcal{L}_{MR}(\mathcal{H})$. Moreover we shall abuse language and also note $\mathcal{L}_B$ and $\mathcal{L}_{MR}$ for the super languages containining all the primitives. As an example, if $\mathcal{H}$ is the subset containing the *get* and *tell* primitives, then the agents of $\mathcal{L}_B(\mathcal{H})$ are those that combine the *get* and *tell* primitives with the sequential, parallel and choice compositional operators. The agents of $\mathcal{L}_{MR}(\mathcal{H})$ are those that combine, with the same compositional operators, elementary agents whose pre and post conditions obey to the correspondance mentioned after definition 4. For instance, ({+t},{+u,-t}) is such an agent of $\mathcal{L}_{MR}(\{tell, get\})$ whereas ({+t,-u},{+u,-t}) is not.

3.1.1 Transition system

To study our two languages, a semantics needs to be defined. To that end, we shall use an operational one in the style of Plotkin ([Plo81]), based on a transition system. The configurations to be considered consist of an agent, summarizing the current state of the agents running

$$(\mathbf{T}) \qquad \langle\, tell(t) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{t\} \,\rangle$$

$$(\mathbf{A}) \quad \langle\, ask(t) \mid \sigma \cup \{t\} \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{t\} \,\rangle$$

$$(\mathbf{G}) \qquad \langle\, get(t) \mid \sigma \cup \{t\} \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle$$

$$(\mathbf{N}) \qquad \frac{t \notin \sigma}{\langle\, nask(t) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle}$$

Figure 3.1: Transition rules for token-based primitives (BachT)

on the store, and a multi-set of tokens, denoting the current state of the store. In order to express the termination of the computation of an agent, we extend the set of agents by adding a special terminating symbol E that can be seen as a completely computed agent. For uniformity purpose, we abuse the language by qualifying E as an agent. To meet the intuition of this terminating agent E, we shall always rewrite agents of the form $(E; A)$, $(E \parallel A)$ and $(A \parallel E)$ as A. This is technically achieved by defining the extended sets of agents as $\mathcal{L}_B \cup \{E\}$ and $\mathcal{L}_{MR} \cup \{E\}$, and by justifying the simplifications by imposing a bimonoid structure.

Figure 3.1 specifies the transition rules for the primitives of the BachT language. The first rule (T) expresses that an atomic agent *tell(t)* can be executed in any store σ, and that its action has the effect of adding the token t to the same store. The second rule (A) states that an atomic agent *ask(t)* can be executed in any store σ containing the token t, however leaving the store σ unaltered after its execution. The third rule (G) works similarly to the previous rule (A), but with the difference of retrieving the token t initially present on the store σ after the execution of the agent *get(t)*. Finally, the fourth rule (N) establishes that an atomic agent *nask(t)* can be executed in any store σ not containing the token t, leaving the store σ unaltered after its execution.

For MRT, the semantics is also defined by a transition system. It turns out that it is possible to define it by one rule. To express it, an auxiliary notion is however needed. It extends the notations of definition 4 to capture the fact that, for each token, the tokens mentioned negatively in the definition are not with their multiplicity on the current store σ.

$$(CM) \qquad \frac{pre^+ \subseteq \sigma,\ pre^- \perp \sigma,\quad \sigma' = (\sigma \setminus post^-) \cup post^+}{\langle (pre, post) \mid \sigma \rangle \longrightarrow \langle E \mid \sigma' \rangle}$$

Figure 3.2: Transition rules for multi-set rewriting-based primitives (MR)

$$(\mathbf{S}) \qquad \frac{\langle A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}{\langle A\ ;\ B \mid \sigma \rangle \longrightarrow \langle A'\ ;\ B \mid \sigma' \rangle}$$

$$(\mathbf{P}) \qquad \frac{\langle A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}{\begin{array}{c}\langle A \mid\mid B \mid \sigma \rangle \longrightarrow \langle A' \mid\mid B \mid \sigma' \rangle \\ \langle B \mid\mid A \mid \sigma \rangle \longrightarrow \langle B \mid\mid A' \mid \sigma' \rangle\end{array}}$$

$$(\mathbf{C}) \qquad \frac{\langle A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}{\begin{array}{c}\langle A\ +\ B \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle \\ \langle B\ +\ A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle\end{array}}$$

Figure 3.3: Transition rules for the operators

Definition 7. For any token t, define $Pre^-[t]$ as the multiset of negatively marked tokens t in the precondition Pre:

$$Pre^-[t] = \{t : -t \in Pre^-\}.$$

Given a precondition Pre and a store σ, we then define the non element-wise inclusion operator $\perp$ as follows:

$$Pre^- \perp \sigma \text{ iff } Pre^-[t] \nsubseteq \sigma, \text{ for any token } t.$$

With this notation, rule (CM) of Figure 3.2 states that a multi-set rewriting $(Pre, Post)$ can be executed in a store σ if the multi-set Pre^+ is included in σ and if no negative pre-condition occurs with the required multiplicity in σ. Under these conditions, the effect of the rewriting is to delete from σ all the negative post-conditions and to add to σ all the positive post-conditions.

Finally, figure 3.3 details the usual rules for sequential composition, parallel composition, interpreted in an interleaved fashion, and CCS-like choice.

The first rule (S) describes the sequential composition of two agents A and B. If the agent A makes a first step to A', transforming the store σ in a store σ', then the global composition moves to A' followed by B, with σ' as resulting store. If A terminates successfully, the composition

is written E ; B which is equal to B. Intuitively, this means that the second agent B will only compute after the full execution of the first agent A. Rule (P) describes the parallel composition of two agents A and B, which is computed in an interleaved way. At every moment one of the two agents may compute, but not both synchronously. After a first step of execution of agent A, both compositions $(A' \parallel B)$ and $(B \parallel A')$ indicate that A' and B must continue their computation in a parallel way. The succesful execution of the global agent $A \parallel B$ is reached when all of its components have finished their own computation. Finally transition rule (C) indicates that a choice between two agents A and B must compute like either A or B , but only one of them, and that the alternative is chosen in view of the first step.

3.1.2 Observables and operational semantics

We are now in a position to define what we want to observe from the computations. Following previous work of the Namur research team on coordination (see eg [BJ99, BJ03a, BJ03b, LJ04, LJ07, LJBB04]), we shall actually take an operational semantics recording the final state of the computations. This is understood as the final store coupled to a mark indicating whether the considered computation is successful or not. Such marks are respectively denoted as δ^+ (for the successful computations) and δ^- (for failed computations). The following definition is valid for BachT and for MRT.

Definition 8.

1. Let δ^+ and δ^- be two fresh symbols denoting respectively success and failure. Define the set of computations $Scomp$ as the cartesian product $Sstore \times \{\delta^+, \delta^-\}$.

2. Define the operational semantics $\mathcal{O} : \mathcal{L}_B \to \mathcal{P}(Scomp)$ as the following function: for any agent $A \in \mathcal{L}_B$

$$\begin{aligned} \mathcal{O}(A) \quad = \quad & \{(\sigma, \delta^+) : \langle A|\varnothing\rangle \to^* \langle E|\sigma\rangle\} \\ & \cup \{(\sigma, \delta^-) : \langle A|\varnothing\rangle \to^* \langle B|\sigma\rangle \nrightarrow, B \neq E\} \end{aligned}$$

   where $\to^*$ denotes the transitive closure of $\to$ and where $\nrightarrow$ denotes the absence of a transition step.

3.2 Expressiveness study

Since the beginning of Computer Science, thousands of programming languages have been invented. They are considered equivalent as they express the same class of functions: the recursive ones. But despite this first property, among them, some are considered to be more

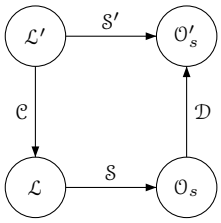


Figure 3.4: Basic embedding.

powerful regarding their capability to express control and data structures. This has induced in the field of sequential languages a line of research, with the goal of formalizing the notion of "power" for a language, with regards to other languages. This "language power" is commonly expressed on the basis of the expressibility or non-expressibility of programming constructs.

In this section, we introduce the notion of modular embedding and use it to compare the different sublanguages of BachT and MRT.

3.2.1 Expressiveness and modular embedding

For comparing the expressiveness of languages, a natural way is to determine whether all programs written in one language can be easily and equivalently translated into the other language. According to this intuition, Shapiro introduced in [Sha92] a first notion of embedding as follows. Consider two languages $\mathcal{L}$ and $\mathcal{L}'$. Assume given the semantic mappings (*Observation criteria*) $\mathcal{S} : \mathcal{L} \to \mathcal{O}_s$ and $\mathcal{S}' : \mathcal{L}' \to \mathcal{O}'_s$, where $\mathcal{O}_s$ and $\mathcal{O}'_s$ are on some suitable domains (typically set of sets of elements). Then $\mathcal{L}$ can *embed* $\mathcal{L}'$ if there exists a mapping $\mathcal{C}$ (coder) from the statements of $\mathcal{L}'$ to the statements of $\mathcal{L}$, and a mapping $\mathcal{D}$ (decoder) from $\mathcal{O}_s$ to $\mathcal{O}'_s$, such that the diagram of Figure 3.4 commutes, namely such that $\mathcal{D}(\mathcal{S}(\mathcal{C}(A))) = \mathcal{S}'(A)$ for every statement $A \in \mathcal{L}'$.

This basic notion of embedding turns out however to be too weak since, for instance, the above equation is satisfied by any pair of Turing-complete languages. De Boer and Palamidessi hence proposed in [dBP94] to add constraints on the coder $\mathcal{C}$ and on the decoder $\mathcal{D}$ in order to obtain a notion of a *modular* embedding usable for concurrent languages. Adopting their work and following previous research in Namur [BJ98, BJ03a, BJ03b, LJ04, LJBB04, LJ07, Lin07], we shall define these constraints as follows:

1. $\mathcal{D}$ should be defined in an element-wise way with respect to $\mathcal{O}_s$, namely for some appropriate mapping $\mathcal{D}_{el}$

$$\mathcal{D}(X) = \{\mathcal{D}_{el}(x) \mid x \in X\} \; for \; any \; X \in \mathcal{O}_s \qquad (P_1)$$

2. the coder $\mathcal{C}$ should be defined in a compositional way with respect to the sequential, parallel and choice operators:

$$\begin{array}{c} \mathcal{C}(A \,;\, B) = \mathcal{C}(A) \,;\, \mathcal{C}(B) \\ \mathcal{C}(A \,||\, B) = \mathcal{C}(A) \,||\, \mathcal{C}(B) \\ \mathcal{C}(A \;+\; B) = \mathcal{C}(A) \;+\; \mathcal{C}(B) \end{array} \qquad (P_2)$$

3. the embedding should preserve the behavior of the original processes with respect to failure and success (*termination invariance*):

$$tm'(\mathcal{D}_{el}(x)) = tm(x) \; for \; any \; X \in \mathcal{O}_s \; and \; x \in X \qquad (P_3)$$

where *tm* and *tm'* extract the termination information (δ^+ or δ^-) from the observables of $\mathcal{L}$ and $\mathcal{L}'$, respectively.

Note that the original formulation of [dBP94] does not impose in P_2 the compositionality property for the sequential operator.

An embedding is then called *modular* if it satisfies properties P_1, P_2, and P_3. The existence of a modular embedding from $\mathcal{L}'$ into $\mathcal{L}$ is subsequently denoted by $\mathcal{L}' \leqslant \mathcal{L}$. It is easy to prove that $\leqslant$ is a pre-order relation. Moreover if $\mathcal{L}' \subseteq \mathcal{L}$ then $\mathcal{L}' \leqslant \mathcal{L}$ that is, any language embeds all its sublanguages. This property descends immediately from the definition of embedding, by setting $\mathcal{C}$ and $\mathcal{D}$ equal to the identity function.

3.2.2 Main results

In [BJ03a, BJ03b, BJ98], Jacquet and Brogi have studied the expressiveness hierarchies of, among others, the different sublanguages of Bach, of a Multi-Set Rewriting language similar to MRT, and the relation between these two families of coordination languages. Figure 3.5 summarizes the separation and equivalence results they obtained in the comparison of the relative expressive power between the different non-trivial sublanguages that can be formed with the primitives *tell*, *ask*, *nask* and *get*. In a similar way, Figure 3.6 summarizes the expressiveness relations they obtained between the different sublanguages of Bach with the sublanguages of the Multi-Set Rewriting formalism.

In these figures, an arrow from language $\mathcal{L}_1$ to language $\mathcal{L}_2$ means that $\mathcal{L}_2$ embeds $\mathcal{L}_1$, that is $\mathcal{L}_1 \leqslant \mathcal{L}_2$. If there is no arrow in the other direction, i.e, from $\mathcal{L}_2$ to $\mathcal{L}_1$ then $\mathcal{L}_1$ is strictly less expressive than $\mathcal{L}_2$, which is noted $\mathcal{L}_1 < \mathcal{L}_2$. Moreover, if $\mathcal{L}_1 \nleqslant \mathcal{L}_2$ and $\mathcal{L}_2 \nleqslant \mathcal{L}_1$ then the

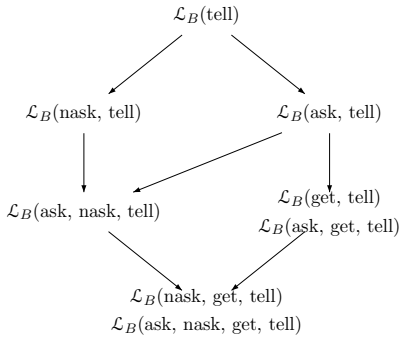


Figure 3.5: Embedding hierarchy of BachT Languages.

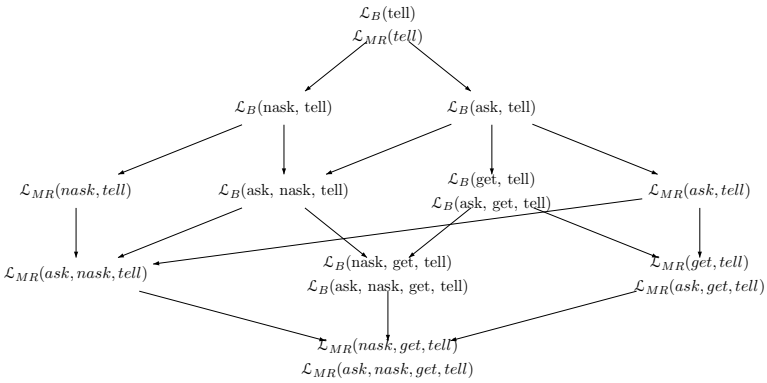


Figure 3.6: Integrated embedding hierarchy of BachT and MRT languages.

languages are incomparable and no relation is drawn in the figures. This is denoted below by $\mathcal{L}_1 \wr \mathcal{L}_2$.

It is worth noting that Figures 3.5 and 3.6 are similar in their structures. This indicates that the hierarchy of the different sublanguages inside the BachT and the MRT families is preserved in the same way. Nevertheless, except for the sublanguages reduced to a *tell* primitive, it is also worth observing that the MRT sublanguages are strictly more expressive than their BachT counterparts. This is due to the capacity of MRT to manipulate atomicaly different tokens.

Expressiveness results will constitute an important part of our work. To provide a background for these results we present in the following subsections several key reasonings used by Jacquet and Brogi in [BJ03a, BJ03b, BJ98]. A complete development is provided in annex (see in sections A.1 and A.2 of Chapter A).

3.2.3 General patterns

We start by pointing out three general patterns. The first one consists in observing that any language embeds its sublanguages.

Generic pattern 1 (Sublanguage inclusion). If $\mathcal{L}' \subseteq \mathcal{L}$ then $\mathcal{L}' \leqslant \mathcal{L}$.

Proof. This property descends immediately from the definition of embedding, by setting $\mathcal{C}$ and $\mathcal{D}$ equal to the identity function. □

Transitivity constitutes a second pattern.

Generic pattern 2 (Transitivity). If $\mathcal{L}_1 \leqslant \mathcal{L}_2$ and $\mathcal{L}_2 \leqslant \mathcal{L}_3$ then $\mathcal{L}_1 \leqslant \mathcal{L}_3$

Proof. Indeed if $\mathcal{C}_{12}$ and $\mathcal{D}_{12}$ denote the coder and decoder for the embedding of $\mathcal{L}_1$ in $\mathcal{L}_2$ and $\mathcal{C}_{23}$ and $\mathcal{D}_{23}$ the coder and decoder for the embedding of $\mathcal{L}_2$ in $\mathcal{L}_3$, then $\mathcal{C}_{23} \circ \mathcal{C}_{12}$ and $\mathcal{D}_{12} \circ \mathcal{D}_{23}$ are the coder and decoder that establish the embedding of $\mathcal{L}_1$ in $\mathcal{L}_3$. □

This property can be used by contraposition to establish the non existence of an embedding.

Generic pattern 3 (Non embedding by transitivity). If $\mathcal{L}_1 \leqslant \mathcal{L}_2$ and $\mathcal{L}_1 \nleqslant \mathcal{L}_3$ then $\mathcal{L}_2 \nleqslant \mathcal{L}_3$

Proof. Indeed, if $\mathcal{L}_2 \leqslant \mathcal{L}_3$ then, thanks to the fact that $\mathcal{L}_1 \leqslant \mathcal{L}_2$, the transitivity pattern leads to the conclusion that $\mathcal{L}_1 \leqslant \mathcal{L}_3$, which contradicts the second hypothesis. □

3.2.4 Expressiveness relations between the BachT sublanguages

Considering the four primitives tell, ask, nask and get, it seems logical to classify the proofs as follows. First we consider the sublanguages that provide tokens in the store with the tell primitive, and that question the store about the presence or absence of tokens on it, respectively with the ask and nask primitives. Then we enrich the language with a get primitive, that permits to retrieve tokens from the store. Finally we consider all the languages that combine the get primitive with the ask and nask primitives.

3.2.4.1 Sublanguages

As a first result, by sublanguage inclusion (pattern 1), a number of modular embeddings are directly established.

Proposition 1. $\mathcal{L}_B(\ \psi) \leqslant \mathcal{L}_B(\ \chi)$, for any subsets of ψ, χ of primitives such that $\psi \subseteq \chi$.

3.2.4.2 Checking for presence and/or absence when adding tokens

The language $\mathcal{L}_B$(tell) is strictly less expressive than $\mathcal{L}_B$(ask,tell).

Proposition 2. $\mathcal{L}_B$(tell) $< \mathcal{L}_B$(ask,tell)

Proof. The proof takes advantage of proposition 1 to establish that $\mathcal{L}_B$(tell) $\leqslant \mathcal{L}_B$(ask,tell) and uses a contradiction technique to establish that $\mathcal{L}_B$(ask,tell) $\not\leqslant \mathcal{L}_B$(tell). Indeed considering the agent $ask(t)$ with $\mathcal{O}(ask(t)) = \{(\varnothing, \delta^-)\}$, a contradiction arises by property P3 of termination invariance, as any agent in $\mathcal{L}_B$(tell) has only successful computations. □

Similarly, the language $\mathcal{L}_B$(tell) is strictly less expressive than $\mathcal{L}_B$(nask,tell).

Proposition 3. $\mathcal{L}_B$(tell) $< \mathcal{L}_B$(nask,tell)

Proof. The technique of the proof is analogous to the one of proposition 2. In particular the contradiction is established by considering the failing behaviour of the agent $tell(t)\ ;\ nask(t)$, whereas any agent in $\mathcal{L}_B$(tell) has only successful computations. □

The two sublanguages $\mathcal{L}_B$(ask,tell) and $\mathcal{L}_B$(nask,tell) are not comparable with each other.

Proposition 4. $\mathcal{L}_B$(ask,tell) $\wr\ \mathcal{L}_B$(nask,tell)

Proof. **(i)** On the one hand, $\mathcal{L}_B$(ask,tell) $\not\leqslant \mathcal{L}_B$(nask,tell) can be established by contradiction by considering the agent $A = tell(t)\ ;\ ask(t)$, for which one has $\mathcal{O}(A) = \{(\{t\}, \delta^+)\}$ and for which it is possible to establish that, for any coder $\mathcal{C}$, the coded agent $\mathcal{C}(A)$ has only failing computations. The proof of this fact is particularly interesting since it evidences the property that, under some hypothesis, failing computations from the empty store can also be repeated from an non-empty store. In our case, as $\mathcal{O}(ask(t)) = \{(\varnothing, \delta^-)\}$, any computation of $\mathcal{C}(ask(t))$ starting on the empty store fails. As $\mathcal{C}(ask(t))$ is composed of nask and tell primitives, this can only occur by having a nask primitive preceded by a tell primitive. As enriching the initial content of the store leads to the same result, any computation starting on any (arbitrary) store fails. As a consequence, even if $\mathcal{C}(tell(t))$ has a successful computation, this computation cannot be continued by a successful computation of $\mathcal{C}(ask(t))$. Consequently any computation of $\mathcal{C}(tell(t); ask(t))$ fails, which produces the announced contradiction.

(ii) On the other hand, $\mathcal{L}_B$(nask,tell) $\not\leqslant \mathcal{L}_B$(ask,tell) is established similarly but, this time, by lifting successful computations on the empty store to successful computations on a non-empty store. More concretely, assume a coder $\mathcal{C}$ and consider $A = tell(t)\ ;\ nask(t)$. One has $\mathcal{O}(A) = \{(\{t\}, \delta^-)\}$. By P_3, the agent $\mathcal{C}(A)$ fails, whereas we shall establish that it has a

successful computation. Indeed, since $\mathcal{O}(tell(t)) = \{(\{t\}, \delta^+)\}$, any computation of $\mathcal{C}(tell(t))$ starting on the empty store is successful. Similarly, it follows from $\mathcal{O}(nask(t)) = \{(\varnothing, \delta^+)\}$ that any computation of $\mathcal{C}(nask(t))$ starting on the empty store is successful. Consequently, so does any computation starting from any store, since $\mathcal{C}(nask(t))$ is composed of ask and tell primitives. Summing up, any (successful) computation of $\mathcal{C}(tell(t))$ starting on the empty store can be continued by a (successful) computation of $\mathcal{C}(nask(t))$, which leads to the announced contradiction. □

It is here worth noting that the reasoning used in the second part of proposition 4 can be used for establishing that $\mathcal{L}_B$(nask,tell) $\not\leqslant$ $\mathcal{L}_B$(get,tell), which is the result of proposition 8. Indeed, replacing ask by get in the $\mathcal{L}_B$(ask,tell) sublanguage does not change the fact that, with $\mathcal{O}(nask(t)) = \{(\varnothing, \delta^+)\}$, any computation of $\mathcal{C}(nask(t))$ is successful starting from any store. This then insures that any successful computation of $\mathcal{C}(tell(t))$ can be followed by a successful computation of $\mathcal{C}(nask(t))$, this leading to an obvious contradiction.

Let us now consider the language $\mathcal{L}(ask, nask, tell)$. The two next following propositions establish that it is strictly more expressive than both $\mathcal{L}(nask, tell)$ and $\mathcal{L}(ask, tell)$.

Proposition 5. $\mathcal{L}_B$(nask,tell) $<$ $\mathcal{L}_B$(ask,nask,tell)

Proof. **(i)** On the one hand, $\mathcal{L}_B$(nask,tell) $\leqslant$ $\mathcal{L}_B$(ask,nask,tell) directly results from proposition 1. **(ii)** On the other hand, $\mathcal{L}_B$(ask,nask,tell) $\not\leqslant$ $\mathcal{L}_B$(nask,tell) is established by pattern 3 on non embedding by transitivity. □

The proof of proposition 6 is established in a similar way to the proof of proposition 5.

Proposition 6. $\mathcal{L}_B$(ask,tell) $<$ $\mathcal{L}_B$(ask,nask,tell)

3.2.4.3 Retrieving tokens from the store

The next important step in the exploration of the different sublanguages is the introduction of the *get* primitive. The first result is that $\mathcal{L}_B$(ask,tell) is strictly less expressive than $\mathcal{L}_B$(get,tell).

Proposition 7. $\mathcal{L}_B$(ask,tell) $<$ $\mathcal{L}_B$(get,tell)

Proof. **(i)** As done in [BJ98], the fact that $\mathcal{L}_B$(ask,tell) $\leqslant$ $\mathcal{L}_B$(get,tell) can be established by translating any $ask(t)$ primitive as $get(t)$; $tell(t)$.

(ii) On the other hand and as done in [BJ98], it is possible to establish that $\mathcal{L}_B$(get,tell) $\not\leqslant$ $\mathcal{L}_B$(ask,tell) by contradictory reasoning. However, differently from [BJ98], this contradiction can be put forward by repeating part of the coding of an agent, while keeping the success of the computation whereas this repetition leads to failure at the non-coded level. More concretely, assume a coder $\mathcal{C}$ and consider $A = tell(t)$; $get(t)$. One has $\mathcal{O}(A) = \{(\varnothing, \delta^+)\}$. By

P_2 and P_3, any computation of $\mathcal{O}(\mathcal{C}(tell(t)) \,;\, \mathcal{C}(get(t))$ is thus successful. Such a computation is composed of a computation for $\mathcal{C}(tell(t))$ followed by a computation for $\mathcal{C}(get(t))$. As $\mathcal{C}(get(t))$ is composed of ask and tell primitives and since ask and tell primitives do not destroy elements, this latter computation can be repeated, which yields successful computations for $\mathcal{O}(\mathcal{C}(tell(t)) \,;\, \mathcal{C}(get(t)) \,;\, \mathcal{C}(get(t)))$. However, $\mathcal{O}(tell(t) \,;\, get(t) \,;\, get(t)) = \{(\varnothing, \delta^-)\}$, which leads to the contradiction. □

It is worth observing that the reasoning used in the second part of proposition 7 can be extended to establish that $\mathcal{L}_B$(get,tell) $\not\leqslant$ $\mathcal{L}_B$(ask,nask,tell), which is part of proposition 10. Indeed, the same agent $A = tell(t) \,;\, get(t)$ has now to be coded not only with primitives *tell* and *ask*, but also with *nask*. Nevertheless, the presence of the *nask* primitive can be dealt with by replacing the sequential composition of get(t) with itself, by a parallel composition, and by mimicking each step of $\mathcal{C}(get(t))$ in the computation of the other instance of $\mathcal{C}(get(t))$. This proof method is also different from the one developped in [BJ98]. This technique will also be used in proposition 27 of section 3.2.5.

The second important result is that $\mathcal{L}_B$(nask,tell) and $\mathcal{L}_B$(get,tell) are not comparable with each other.

Proposition 8. $\mathcal{L}_B$(nask,tell) $\wr$ $\mathcal{L}_B$(get,tell)

Proof. The proposition directly results, on the one hand, from the same reasoning employed in the second part of proposition 4 and, on the other hand, from pattern 3 coupled to proposition 4. □

Without big surprise, by coding any ask primitive as a sequence of get followed by tell, the sublanguages $\mathcal{L}_B$(get,tell) and $\mathcal{L}_B$(ask,get,tell) appear to be equivalent.

Proposition 9. $\mathcal{L}_B$(get,tell) = $\mathcal{L}_B$(ask,get,tell)

The next proposition establishes that $\mathcal{L}_B$(get,tell) is not comparable with $\mathcal{L}_B$(ask,nask,tell).

Proposition 10. $\mathcal{L}_B$(ask,nask,tell) $\wr$ $\mathcal{L}_B$(get,tell)

Proof. **(i)** On the one hand, $\mathcal{L}_B$(ask,nask,tell) $\not\leqslant$ $\mathcal{L}_B$(get,tell) is established, as in [BJ98], by pattern 3 of non embedding by transitivity.

(ii) On the other hand, $\mathcal{L}_B$(get,tell) $\not\leqslant$ $\mathcal{L}_B$(ask,nask,tell) may be established by contradiction, similarly to the second part of proposition 7, by replacing the sequential composition of the two $get(t)$ primitives by a parallel one, in order to cope with the potential presence of nask primitives. As the careful reader will have noticed, this proof technique is different from the one used in [BJ98]. □

3.2.4.4 Checking for presence and/or absence when adding and/or retrieving tokens

The first observation is that the introduction of `nask` to $\mathcal{L}_B$(get,tell) increases the expressive power of the new sublanguage. The proof methods of both parts are similar to those employed previously and are thus not repeated here.

Proposition 11. $\mathcal{L}_B(\text{get,tell}) < \mathcal{L}_B(\text{nask,get,tell})$

$\mathcal{L}_B$(nask,get,tell) and $\mathcal{L}_B$(ask,nask,get,tell) are equivalent. One of the proof method used in [BJ98] to establish this result are worth stressing here. They highlight the fact that a token can be coded by two tokens, thanks to the enumerably property of tokens.

Proposition 12. $\mathcal{L}_B(\text{nask,get,tell}) = \mathcal{L}_B(\text{ask,nask,get,tell})$

Proof. **(i)** On the one hand, $\mathcal{L}_B(\text{nask,get,tell}) \leqslant \mathcal{L}_B(\text{ask,nask,get,tell})$ is established by proposition 1 on language inclusion.

(ii) On the other hand, to establish $\mathcal{L}_B(\text{ask,nask,get,tell}) \leqslant \mathcal{L}_B(\text{nask,get,tell})$ we shall provide a coder such that the coding of the primitives $ask(t)$ and $nask(t)$ manipulate different tokens.

As the set of tokens is denumerable, it is possible [1] to associate each of them, say t, to a pair (t_1, t_2). Given such a coding of tokens, we define the compositional coder $\mathcal{C}$ as follows:

$$\begin{array}{lcl} \mathcal{C}(ask(t)) & = & get(t_2)\ ;\ tell(t_2) \\ \mathcal{C}(nask(t)) & = & nask(t_1) \\ \mathcal{C}(get(t)) & = & get(t_2)\ ;\ get(t_1) \\ \mathcal{C}(tell(t)) & = & tell(t_1)\ ;\ tell(t_2) \end{array}$$

The decoder $\mathcal{D}$ is defined as follows: $\mathcal{D}_{el}((\sigma, \delta)) = (\overline{\sigma}, \delta)$, where $\overline{\sigma}$ is composed of the tokens t for which t_1 and t_2 are in σ, the multiplicity of t being that of pairs (t_1, t_2) in σ.

With those definitions of the coder $\mathcal{C}$ and the decoder $\mathcal{D}$, properties P1 of element-wise and P2 of compositionality are guaranteed. It remains to establish property P3 of termination invariance and that $\mathcal{O}(A) = \mathcal{D}(\mathcal{O}(\mathcal{C}(A)))$ for any agent A of $\mathcal{L}_B$(ask,nask,get,tell). The proof consists of establishing that for any agent A and stores σ and τ:

1. $\langle A|\overline{\sigma}\rangle \rightarrow^* \langle E|\overline{\tau}\rangle$ iff $\langle \mathcal{C}(A)|\sigma\rangle \rightarrow^* \langle E|\tau\rangle$
2. there is some agent B such that $\langle A|\overline{\sigma}\rangle \rightarrow^* \langle B|\overline{\tau}\rangle \nrightarrow$ with $B \neq E$
   iff there is some agent $B' \in \mathcal{L}_B$(nask,get,tell) such that $\langle \mathcal{C}(A)|\sigma\rangle \rightarrow^* \langle B'|\tau\rangle \nrightarrow$

This is proved by induction on the structure of the agent.

[1] for instance it suffices to associate the token associated with the integer n to the tokens associated with the integers $2n$ and $2n+1$

□

Finally, as in [BJ98], it is possible to prove that $\mathcal{L}_B$(nask,get,tell) is strictly less expressive than $\mathcal{L}_B$(ask,nask,get,tell). The proof does not highlight any new reasoning and is thus not repeated here.

Proposition 13. $\mathcal{L}_B$(ask,nask,tell) $<$ $\mathcal{L}_B$(ask,nask,get,tell)

3.2.5 BachT in comparison with MRT

In [BJ99, BJ03b], BachT has also been compared to a multi-set rewriting language (MRT). However, it is worth observing that the version of MRT that we consider in this thesis is actually more expressive than the one studied by Brogi and Jacquet. Indeed, in their work, a negatively annotated token t in a pre-condition is seen as stating that t is absent from the store. Under that interpretation, it makes a priori no sense to consider two or more occurrences of t negatively marked. However, our work on multiplicity of tokens sheds a new light for a new interpretation. Indeed, as introduced in definition 7, specifying n occurrences of $-t$ in a pre-condition allows t to appear on the store but with a multiplicity strictly less than n. The interpretation used by Jacquet and Brogi follows as a particular case by using $n = 1$. However, no counterpart for $n > 1$ can be found in [BJ99, BJ03b], which leads our language variant to be strictly more powerful.

Nevertheless, the techniques used in [BJ99, BJ03b] to establish the expressiveness results can also be directly reused to establish the same expressiveness results for our version of MRT. Following what we did for BachT, we will only subsequently sketch the main reasonings. The interested reader is referred to appendix A where all the expressiveness results are established, evidencing thereby that the results of [BJ99, BJ03b] transpose to our MRT language.

In these lines, as a complement to the three patterns listed in subsection 3.2.2, two additional patterns are worth being made explicit. Before presenting them, it is first necessary to define what we shall call a *normal form*. As pointed out among others in [BJ98, BJ99, Lin07], a classical result of concurrency theory is that modeling parallel composition by interleaving, as we do, allows agents to be considered in a normal form. We first define what this actually means, and then state the proposition that agents and their normal forms are equivalent in the sense that they yield the same computations.

Definition 9. The set *Snagent* of agents in *normal form* is defined by the following rule, where N is an agent in normal form, C denotes a communication action as defined by syntactic rules

in definitions 2 and 5 for BachT and definitions 3 and 6 for MRT and A is an agent as defined in these definitions.

$$N ::= C \mid C \ ; A \mid N \ + \ N$$

This normal form is similar to the one proposed in the expansion theorem of Milner. Note that the main idea of the normal form is to make explicit the first primitive steps as well as the choices for these first steps. It is also worth observing that parallel composition does not appear explicitly in the grammar rule of definition 9. In fact it can only appear through the agent A. As a result, $tell(t) \mid\mid tell(u)$ has to be rewritten as $tell(t) \ ; tell(u) \ + \ tell(u) \ ; tell(t)$, which has indeed the effect of pointing out the choices of primitives steps to be made in the computation of $tell(t) \mid\mid tell(u)$. Other examples of normal forms are

$$tell(t) \ ; (ask(u) \ + \ get(v))$$

and

$$tell(t) \ ; get(u) \ + \ nask(v) \ ; (tell(w) \ + \ tell(u))$$

Proposition 14. For any agent A of BachT or MRT there is an agent in normal form N such that $\mathcal{O}(\mathrm{N}) = \mathcal{O}(\mathrm{A})$.

Generic pattern 4 (Presence). In this pattern, the coded version of an agent expressed in a multi-set rewrited form is written in a normal form. The key point of the demonstration consists in proving that no alternative in this normal form exists, reducing the coder to an empty statement. This is in contradiction with definition 6 which forces agents to contain at least one primitive and thus to be non empty.

Generic pattern 5 (Absence). In this pattern, the coded version of any token results in the association of a finite set of tokens. The proof works by contradiction. It supposes the existence of a coder $\mathcal{C}$, and associates to any tokens x and y their coded images, consisting in the finite sets of tokens S_x and S_y respectively. The key point of the demonstration establishes on the one hand that either there exist two tokens a and b such that $S_a \cap S_b = \varnothing$, in which case a contradiction can be produced directly, or that there is a token a for which any other token b is such that $S_a \cap S_b \neq \varnothing$. In that case, a series of tokens x_i can be constructed, out of which a contradiction arises.

In a way similar to the presentation of the main results of expressiveness relation between the different sublanguages of BachT, the results of the comparison between BachT and the multi-set rewriting language MRT are subsequently grouped together according to a logical introduction of the different primitives. The first step consists in allowing to place tokens on

the store, with the tell primitive. The second step introduces ask and nask primitives, allowing to question the store about the presence or absence of tokens on it. The third step introduces the get primitive, for retrieving tokens from the store. Finally, the last propositions tackle all the primitives. As before, we shall only detail proofs which contain interesting techniques and refer the reader to annex A for a complete set of proofs.

3.2.5.1 Sublanguages

As a first result, in view of the correspondence between MRT primitives and their BachT counterparts (see section 3.1, definition 4 and consequence), it is possible to establish that any sublanguage of BachT is embedded in its correspondig sublanguage in MRT.

Proposition 15. $\mathcal{L}_B(\ \chi) \leqslant \mathcal{L}_{MR}(\ \chi)$, for any subset of χ of primitives.

3.2.5.2 Putting tokens on the store

Proposition 16. $\mathcal{L}_B$(tell) and $\mathcal{L}_{MR}$(tell) are equivalent.

As a result of the expressiveness hierarchy studied in the previous section, and illustrated in figure 3.5, it follows that both languages $\mathcal{L}_B$(ask,tell) and $\mathcal{L}_B$(nask,tell) are strictly more expressive than $\mathcal{L}_{MR}$(tell) since both have been established strictly more expressive than $\mathcal{L}_B$(tell).

Let us now introduce the ask and nask primitives. The next section compares the different sublanguages between the BachT language and the multi-set rewriting.

3.2.5.3 Checking for presence and/or absence when adding tokens

The first proposition compares $\mathcal{L}_B$(ask,tell) with its multiset counterpart.

Proposition 17. $\mathcal{L}_B$(ask,tell) $< \mathcal{L}_{MR}$(ask,tell)

Proof technique 1. This method of demonstration is used to prove other propositions as well, namely the propositions 22, 23, 24 and 30 respectively demonstrating that $\mathcal{L}_{MR}$(ask,tell) $\nleqslant$ $\mathcal{L}_B$(get,tell), $\mathcal{L}_{MR}$(ask,tell) $\nleqslant$ $\mathcal{L}_B$(nask,get,tell), $\mathcal{L}_{MR}$(ask,tell) $\nleqslant$ $\mathcal{L}_B$(ask,nask,tell) and $\mathcal{L}_{MR}$(get,tell) $\nleqslant$ $\mathcal{L}_B$(get,tell). Indeed, in each case the intuition is that every BachT language in those propositions is unable to atomically test the presence of two tokens. The demonstration follows the same steps: an agent is expressed in a multi-set way, that expresses the simultaneous

testing of presence, or simultaneous retrieve of some tokens. The coded version of these agents are expressed in a normal form and the technique of demonstration consists in establishing by contradiction that none of the alternatives in these normal forms exist, which is impossible since the coded version must contain at least one primitive, following definition 6. This method inscribes itself in the pattern 4 of presence.

More concretely, given that $\mathcal{L}_B(\text{ask,tell}) \leqslant \mathcal{L}_{MR}(\text{ask,tell})$ follows directly from proposition 15, let us establish that $\mathcal{L}_{MR}(\text{ask,tell}) \nleqslant \mathcal{L}_B(\text{ask,tell})$ by contradiction. Consider $AB = (\{+a, +b\}, \{\})$ and assumes that $\mathcal{C}(AB)$ is in normal form (see definition 9) and thus is written as

$$tell(t_1); A_1 + \cdots + tell(t_p); A_p + ask(u_1); B_1 + \cdots + ask(u_q); B_q$$

In this expression, we will establish that there is no alternative guarded by a $tell(t_i)$ operation and no alternative guarded by an $ask(u_j)$ operation either, which is impossible since $\mathcal{C}(AB)$ must contain at least one primitive.

Let us first establish by contradiction that there is no alternative guarded by a $tell(t_i)$ operation. Indeed, if there is an alternative guarded, say by $tell(t_i)$, then

$$D = \langle \mathcal{C}(AB) | \varnothing \rangle \rightarrow \langle A_i | t_i \rangle$$

is a valid computation prefix of $\mathcal{C}(AB)$. It should deadlock afterwards since $\mathcal{O}(AB) = (\varnothing, \delta^-)$. However D is also a valid computation prefix of $\mathcal{C}(AB + (\{\}, \{+a\}))$. Hence, $\mathcal{C}(AB + (\{\}, \{+a\}))$ admits a failing computation which contradicts the fact that $\mathcal{O}(AB + (\{\}, \{+a\})) = \{(\{a\}, \delta^+)\}$.

Secondly, we establish that there is also no alternative guarded by an $ask(u_j)$ operation. To that end, let us first consider two auxiliary computations. As $\mathcal{O}((\{\}, \{+a\})) = \{(\{a\}, \delta^+)\}$, any computation of $\mathcal{C}((\{\}, \{+a\})$ starting in the empty store succeeds. Let

$$\langle \mathcal{C}((\{\}, \{+a\})) | \varnothing \rangle \rightarrow \cdots \rightarrow \langle E | \{a_1, \ldots, a_m\} \rangle$$

be such a computation. Similarly, let

$$\langle \mathcal{C}((\{\}, \{+b\})) | \varnothing \rangle \rightarrow \cdots \rightarrow \langle E | \{b_1, \ldots, b_n\} \rangle$$

be one computation of $\mathcal{C}((\{\}, \{+b\}))$. The proof of the claim proceeds by first establishing that none of the u_i's belong to $\{a_1, \ldots, a_m\} \cup \{b_1, \ldots, b_n\}$.

First let us prove that none of the u_j's belong to $\{a_1, \ldots, a_m\}$. By contradiction, assume that $u_i = a_k$ for some k. Then

$$D' = \langle \mathcal{C}((\{\}, \{+a\}); AB) | \varnothing \rangle \rightarrow \cdots \rightarrow \langle AB | \{a_1, \ldots, a_m\} \rangle \rightarrow \langle B_j | \{a_1, \ldots, a_m\} \rangle$$

is a valid computation prefix of $\mathcal{C}((\{\}, \{+a\}); AB)$, which can only be continued by failing suffixes. However D' induces the following computation prefix D'' for $(\{\}, \{+a\}); (AB + (\{+a\}, \{\}))$

which as just seen admits only successful computations:

$$D'' = \langle \mathcal{C}((\{\},\{+a\});(AB + (\{+a\},\{\})))|\varnothing\rangle \rightarrow \ldots$$
$$\rightarrow \langle AB + (\{+a\},\{\})|\{a_1,\ldots,a_m\}\rangle \rightarrow \langle B_j|\{a_1,\ldots,a_m\}\rangle$$

The proof proceeds similarly in the case some $u_j \in \{b_1,\ldots,b_n\}$ for some $j \in \{1,\ldots,q\}$ by then considering $(\{\},\{+b\});AB$ and $(\{\},\{+b\});(AB + (\{+b\},\{\}))$.

Finally, the fact that the $u'_j s$ do not belong to $\{a_1,\ldots,a_m\} \cup \{b_1,\ldots,b_n\}$ induces a contradiction. Indeed, if this is the case, then

$$\langle \mathcal{C}((\{\},\{+a\});(\{\},\{+b\});AB)|\varnothing\rangle \rightarrow \ldots$$
$$\rightarrow \langle (\{\},\{+b\});AB)|\{a_1,\ldots,a_m\}\rangle \rightarrow$$
$$\cdots \rightarrow \langle AB|\{a_1,\ldots,a_m,b_1,\ldots,b_n\}\rangle \nrightarrow$$

is a valid failing computation prefix of $\mathcal{C}((\{\},\{+a\});(\{\},\{+b\});AB)$ whereas $(\{\},\{+a\});(\{\},\{+b\});AB$ has only one successful computation.

Symmetrically to proposition 17, $\mathcal{L}_B$(nask,tell) is strictly less expressive than $\mathcal{L}_{MR}$(nask,tell). In order to establish this result, we first introduce the following lemma.

Lemma 1. Let $f : Stoken \rightarrow \mathcal{P}_f(Stoken)$ be a function associating each token with a finite set of tokens. Assume that $f(a) \cap f(b) \neq \varnothing$, for any pair of distinct tokens a and b. Then there is a denumerable sequence of distinct tokens x_i's and an integer N such that

$$\bigcap_{i=1}^{N} f(x_i) \neq \varnothing$$

and

$$\bigcap_{i=1}^{N} f(x_i) = \bigcap_{i=1}^{N} f(x_i) \cap f(x_j)$$

for any $j > N$.

Proof. The proof of this technical lemma has been established in [BJ03b], to which we refer the reader. The intuition of the lemma is as follows. For any token t handled in a primitive, the execution of the coding of the associated tell(t) primitive induces a coding of the token, in the form of a finite set of tokens. If for any pair of distinct tokens, the intersection of the induced codings is not empty, then for a denumerable sequence of distinct tokens, only a subsequence will share a common intersection, while the remaining tokens of the sequence will share the same intersection. □

Let us now establish that $\mathcal{L}_B$(nask,tell) is strictly less expressive than $\mathcal{L}_{MR}$(nask,tell).

Proposition 18. $\mathcal{L}_B$(nask,tell) $< \mathcal{L}_{MR}$(nask,tell).

Proof technique 2. This method of demonstration is also used to prove other propositions, namely the propositions 21, and 26, where we respectively demonstrate that $\mathcal{L}_{MR}$(nask,tell) $\nleqslant$ $\mathcal{L}_B$(ask,nask,tell) and $\mathcal{L}_{MR}$(nask,tell) $\nleqslant$ $\mathcal{L}_B$(nask,get,tell). The proof follows the same two steps: on the one hand it considers that the codings S_a and S_b of two tokens a and b share no elements, and on the other hand, it considers the contrary, i.e. that the codings share at least one common element. In both cases, those two hypotheses produce a contradiction for a certain agent, reducing the coding of this last one to an empty statement. Note that for proposition 21 the presence of the nask primitives in the considered BachT languages imposes to use the parallel operator to build the coding of some primitives. This corollary inscribes itself in pattern 5 of absence.

More concretely, given that $\mathcal{L}_B$(nask,tell) $\leqslant$ $\mathcal{L}_{MR}$(nask,tell) holds by proposition 15, let us establish that $\mathcal{L}_{MR}$(nask,tell) $\nleqslant$ $\mathcal{L}_B$(nask,tell) by assuming the existence of a coder $\mathcal{C}$, and by establishing that it contains in fact no primitive, while it has to contain at least one.

The proof is similar to part (ii) of proposition 17, but this time by exploiting the inability of $\mathcal{L}_B$(nask,tell) to atomically test the absence of two distinct tokens a and b, following the schema of pattern 5 of absence.

To do so the construction of the tokens $\{a_1, \ldots, a_m\}$ and $\{b_1, \ldots, b_n\}$ associated with the coding of a and b is generalized by the definition of a function $f : Stoken \rightarrow \mathcal{P}_f(Stoken)$, associating with each token a finite set of tokens.

For any token t, as $\mathcal{O}((\{\}, \{+t\})) = \{(\{t\}, \delta^+)\}$, any computation of $\mathcal{C}((\{\}, \{+t\}))$ starting in the empty store succeeds. Let $\langle((\{\}, \{+t\}))|\varnothing\rangle \rightarrow \cdots \rightarrow \langle E|\{t_1, \ldots, t_{m_t}\}\rangle$ be such a computation and let S_t denote the resulting store $\{t_1, \cdots, t_{m_t}\}$.

Then the proof of the claim proceeds by examining two cases: (I) either there exist two (distinct) tokens a and b such that $S_a \cap S_b = \varnothing$, (II) or $S_a \cap S_b \neq \varnothing$ for any pair of (distinct) tokens a and b.

CASE I: Let us first suppose that there are two tokens a and b such that $S_a \cap S_b = \varnothing$. One considers $AB = (\{-a, -b\}, \{\})$ and $\mathcal{C}(AB)$ in its normal form:

$$tell(v_1)\ ;\ A_1 + \cdots + tell(v_p)\ ;\ A_p + nask(u_1)\ ;\ B_1 + \cdots + nask(u_q)\ ;\ B_q$$

The proof then proceeds by establishing that there are no alternatives guarded by $tell(v_i)$ nor by $nask(u_j)$. The absence of alternative guarded by a $tell(v_i)$ primitive is established as in part (ii) of proposition 17: if this was not the case, then AB would point out a deadlocking computation for $(\{\}, \{+a\}); (AB + (\{\}, \{+a\}))$ which only admits successful computations. To prove the absence of alternatives guarded by a $nask(u_j)$ primitive, one establishes that the u_j's should belong to S_a and to S_b, which is impossible since $S_a \cap S_b = \varnothing$. By contradiction, assume

that $u_j \notin S_a$ for some j (the case where $u_j \notin S_b$ is treated similarly). Then

$$\langle \mathcal{C}((\{\},\{+a\})\ ;\ AB) \mid \varnothing\rangle \longrightarrow \cdots \longrightarrow \langle \mathcal{C}(AB) \mid S_a\rangle \longrightarrow \langle B_j \mid S_a\rangle$$

is a valid computation prefix of $\mathcal{C}((\{\},\{+a\})\ ;\ AB)$ which can only be continued by failing suffixes. However, this prefix induces the following computation prefix D' for $\mathcal{C}((\{\},\{+a\})\ ;\ (AB + (\{\},\{+a\}))$ which should only admit successful computations:

$$\langle \mathcal{C}((\{\},\{+a\})\ ;\ (AB + (\{\},\{+b\}))) \mid \varnothing\rangle \longrightarrow \cdots$$

$$\longrightarrow \langle AB + (\{\},\{+b\}) \mid S_a\rangle \longrightarrow \langle B_j \mid S_a\rangle$$

CASE II: Let us now suppose that $S_a \cap S_b \neq \varnothing$ for any pair of tokens a and b. As proved by Lemma 1, it is possible to construct an infinite sequence of distinct tokens x_i's and to identify an integer n such that

$$\bigcap_{i=1}^{n} S_{x_i} \neq \varnothing$$

and

$$\bigcap_{i=1}^{n} S_{x_i} = \bigcap_{i=1}^{n} S_{x_i} \cap S_{x_j}$$

for any $j > n$. Let us consider $NT = (\{-x_1, \cdots, -x_n\}, \{\})$ and $\mathcal{C}(NT)$ in its normal form

$$tell(v_1)\ ;\ A_1 + \cdots + tell(v_p)\ ;\ A_p + nask(u_1)\ ;\ B_1 + \cdots + nask(u_q)\ ;\ B_q$$

By using a reasoning similar to the one employed for case I, one may prove that there are no alternatives guarded by a $tell(v_i)$ primitive and that $\{u_1, \cdots, u_q\} \subseteq S_{x_1} \cap \cdots \cap S_{x_n}$. Therefore $\mathcal{C}((\{\},\{+x_{n+1}\})\ ;\ NT)$ has a failing computation since $S_{x_1} \cap \cdots \cap S_{x_n} \cap S_{x_{n+1}} = S_{x_1} \cap \cdots \cap S_{x_n}$ and thus $\{u_1, \cdots, u_q\} \subseteq S_{x_1} \cap \cdots \cap S_{x_n} \subseteq S_{x_{n+1}}$. However, this contradicts the fact that $(\{\},\{+x_{n+1}\})\ ;\ NT$ has only one successful computation.

In conclusion, $\mathcal{C}(AB)$ reduces to an empty statement, which is not possible since it should contain at least one primitive.

$\mathcal{L}_{MR}$(ask,tell) and $\mathcal{L}_{B}$(nask,tell) are not comparable with each other and so are $\mathcal{L}_{MR}$(nask,tell) and $\mathcal{L}_{B}$(ask,tell).

Proposition 19. $\mathcal{L}_{MR}$(ask,tell) $\wr$ $\mathcal{L}_{B}$(nask,tell)

Proposition 20. $\mathcal{L}_{MR}$(nask,tell) $\wr$ $\mathcal{L}_{B}$(ask,tell)

The fact that $\mathcal{L}_{B}$(ask,nask,tell) and $\mathcal{L}_{MR}$(nask,tell) are not comparable with each other can be established by using the same proof technique as for proposition 18.

Proposition 21. $\mathcal{L}_{B}$(ask,nask,tell) $\wr$ $\mathcal{L}_{MR}$(nask,tell)

3.2.5.4 Retrieving tokens from the store in the BachT language

By using previous reasoning patterns, the sublanguage $\mathcal{L}_B$(get,tell) is not comparable with the sublanguage $\mathcal{L}_{MR}$(ask,tell).

Proposition 22. $\mathcal{L}_B$(get,tell) $\wr$ $\mathcal{L}_{MR}$(ask,tell)

The language $\mathcal{L}_{MR}$(ask,tell) turns out to be incomparable with $\mathcal{L}_B$(nask,get,tell). The proof of this fact uses a saturation technique that is worth exploring for further results.

Proposition 23. $\mathcal{L}_B$(nask,get,tell) $\wr$ $\mathcal{L}_{MR}$(ask,tell)

Proof. **(i)** On the one hand, $\mathcal{L}_B$(nask,get,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(ask,tell). Otherwise, by pattern 3 of non embedding by transitivity, $\mathcal{L}_B$(nask,tell) $\leqslant$ $\mathcal{L}_{MR}$(ask,tell) which has been proved impossible in part (ii) of proposition 19.

(ii) On the other hand, $\mathcal{L}_{MR}$(ask,tell) $\not\leqslant$ $\mathcal{L}_B$(nask,get,tell) is established as in [BJ99, BJ03b]. The intuition behind the proof is again that $\mathcal{L}_B$(nask,get,tell) is not able to test atomically the presence of two distinct tokens a and b. Following pattern 4 of presence, we then proceed by contradiction using these two tokens a and b. However, the destructive character of *get* primitives coupled to the test for absence of *nask* slightly complicate our task of producing a contradiction. To that end, we shall "saturate" their effect by taking as many instances of codings in parallel and thereby by extending the sets S_b introduced in part (ii) of the proof of proposition 18.

Let us thus proceed by contradiction by assuming the existence of a coder $\mathcal{C}$. Take two distinct tokens a and b. Let n be the number of occurences of *nask* and *get* primitives of $\mathcal{C}((\{\},\{+a\}))$. As $\mathcal{C}((\{\},\{+a\}))$ has only successful computations, let, as in the part (ii) of the proof of proposition 18, S_a be the store resulting from one of them. As $(||_{k=1}^{n+2}(\{\},\{+b\}))$; $(\{\},\{+a\}))$ succeeds as well, let S'_b denote the store resulting from one successful computation of its coding. Consider finally $ABs = (\{+a,+b,\cdots,+b\},\{\})$ requesting one a with $n+3$ copies of b and $\mathcal{C}(ABs)$ in its normal form:

$$\begin{array}{l} tell(t_1)\ ;\ A_1 + \cdots + tell(t_p)\ ;\ A_p \\ + get(u_1)\ ;\ B_1 + \cdots + get(u_q)\ ;\ B_q \\ + nask(v_1)\ ;\ C_1 + \cdots + nask(v_r)\ ;\ C_r \end{array}$$

Following proof technique 1, we shall establish (I) that there are no alternatives guarded by *tell* and *nask* primitives, and (II) that $\{u_1,\cdots,u_q\} \cap (S_a \cup S'_b) = \varnothing$. Assuming these two points proved, a contradiction can be produced as follows. In view of the saturation provided by the $n+2$ copies of $\mathcal{C}((\{\},\{+b\}))$, adding one more only adds tokens present in $S_a \cup S'_b$. As a result, $\mathcal{C}(||_{k=1}^{n+3}(\{\},\{+b\}))$; $(\{\},\{+a\})$; $ABs)$ fails whereas $||_{k=1}^{n+3}(\{\},\{+b\}))$; $(\{\},\{+a\})$; ABs has only one successful computation. Hence the contradiction.

STEP I: Let us first establish that there are no alternative guarded by a $tell(t_i)$ primitive. The proof proceeds by contradiction as in part (ii) of the proof of proposition 17, by pointing out a failing computation for $\mathcal{C}(AB + (\{\}, \{+a\}))$, contradicting the fact that $\mathcal{O}(AB + (\{\}, \{+a\})) = \{(\{a\}, \delta^+)\}$.

In a similar way there are no alternative guarded by a *nask* primitive. Indeed assuming the existence of a $nask(v_i)$; C_i alternative again points out a failing computation for $\mathcal{C}(AB + (\{\}, \{+a\}))$, contradicting the fact that $\mathcal{O}(AB + (\{\}, \{+a\})) = \{(\{a\}, \delta^+)\}$.

STEP II: Let us now establish that $\{u_1, \cdots, u_q\} \cap (S_a \cup S'_b) = \varnothing$. This is proved in two steps by establishing (1) that $\{u_1, \cdots, u_q\} \cap S_a = \varnothing$, and (2) that $\{u_1, \cdots, u_q\} \cap S'_b = \varnothing$.

First we have that $\{u_1, \cdots, u_q\} \cap S_a = \varnothing$. By contradiction, assume that $u_i \in S_a$ for some i. Let us observe that each step of the considered computation of $\mathcal{C}((\{\}, \{+a\}))$ can be repeated in turn, in as many parallel occurences of it as needed, so that

$$\begin{aligned} P = \langle \mathcal{C}((||_{k=1}^{q}(\{\}, \{+a\})) \ ; \ ABs) | \varnothing \rangle \\ \rightarrow \cdots \rightarrow \ \langle ABs | \cup_{k=1}^{q} S_a \rangle \\ \rightarrow \quad \langle B_i | (\cup_{k=1}^{q} S_a) \backslash \{\overline{u_i}\} \rangle \end{aligned}$$

is a valid computation prefix of $\mathcal{C}((||_{k=1}^{q}(\{\}, \{+a\})) \ ; \ ABs)$, which can only be continued by failing suffixes. However P induces the following computation prefix P' for $\mathcal{C}((||_{k=1}^{q}(\{\}, \{+a\})) \ ; \ (ABs + (\{\}, \{+a\})))$ which admits only successful computations:

$$\begin{aligned} P' = \langle \mathcal{C}((||_{k=1}^{q}(\{\}, \{+a\})) \ ; \ (ABs + (\{\}, \{+a\}))) | \varnothing \rangle \\ \rightarrow \cdots \rightarrow \ \langle \mathcal{C}(ABs + (\{\}, \{+a\})) | \cup_{k=1}^{q} S_a \rangle \\ \rightarrow \quad \langle B_i | (\cup_{k=1}^{q} S_a) \backslash \{\overline{u_i}\} \rangle \end{aligned}$$

Hence the contradiction.

The fact that $\{u_1, \cdots, u_q\} \cap S'_b = \varnothing$ is proved similarly, by considering S'_b instead of S_a and $(\{\}, \{+b\})$ instead of $(\{\}, \{+a\})$. □

By using similar reasonings as for the above proposition, it is possible to establish that $\mathcal{L}_{MR}$(ask,tell) is not comparable with $\mathcal{L}_B$(ask,nask,tell).

Proposition 24. $\mathcal{L}_{MR}$(ask,tell) $\wr$ $\mathcal{L}_B$(ask,nask,tell)

Similarly, by using previous reasonings, it is possible to establish that $\mathcal{L}_B$(ask,nask,tell) is stricly less expressive than $\mathcal{L}_{MR}$(ask,nask,tell).

Proposition 25. $\mathcal{L}_B$(ask,nask,tell) $< \mathcal{L}_{MR}$(ask,nask,tell)

Additional incomparability results can be established using the reasonings exposed previously. The first one requires to extend lemma 1 as follows.

Lemma 2. Let S be a finite set of tokens. Let $f : Stoken \rightarrow \mathcal{P}_f(Stoken)$ be a function associating to each token a finite set of tokens. Assume that $S \cap f(x) \neq \varnothing$, for any token x. Then there is a denumerable sequence of distinct tokens x_i's and an integer N such that

$$\bigcap_{i=1}^{N}(S \cap S_{x_i}) \neq \varnothing$$

and

$$\bigcap_{i=1}^{N}(S \cap S_{x_i}) = \bigcap_{i=1}^{N}(S \cap S_{x_i}) \cap S_{x_j}$$

for any $j > N$. In particular, $(\bigcap_{i=1}^{N}(S \cap S_{x_i})) \cap (S \cap S_{x_j}) \neq \varnothing$, for any $j > N$.

Using this lemma and similar proofs as before, it is possible to establish the following results.

Proposition 26. $\mathcal{L}_B$(nask,get,tell) $\wr$ $\mathcal{L}_{MR}$(nask,tell)

Proposition 27. $\mathcal{L}_{MR}$(ask,nask,tell) $\wr$ $\mathcal{L}_B$(nask,get,tell)

Proposition 28. $\mathcal{L}_B$(get,tell) $\wr$ $\mathcal{L}_{MR}$(nask,tell)

Proposition 29. $\mathcal{L}_B$(get,tell) $\wr$ $\mathcal{L}_{MR}$(ask,nask,tell)

3.2.5.5 Retrieving tokens from the store in MRT

Previous reasonings allow to establish the following propositions.

Proposition 30. $\mathcal{L}_B$(get,tell) $< \mathcal{L}_{MR}$(get,tell)

Proposition 31. $\mathcal{L}_{MR}$(get,tell) $\wr$ $\mathcal{L}_B$(nask,tell)

Proposition 32. $\mathcal{L}_{MR}$(get,tell) $\wr$ $\mathcal{L}_B$(nask,get,tell)

Proposition 33. $\mathcal{L}_{MR}$(get,tell) $\wr$ $\mathcal{L}_B$(ask,nask,tell)

3.2.5.6 Checking for presence and/or absence when adding and/or retrieving tokens

Finally, it is possible to establish that $\mathcal{L}_B$(ask,nask,get,tell) is strictly less expressive than $\mathcal{L}_{MR}$(ask,nask,get,tell). The proof illustrates the coding of the BachT primitives in the MRT ones.

Proposition 34. $\mathcal{L}_B$(ask,nask,get,tell) $< \mathcal{L}_{MR}$(ask,nask,get,tell)

Proof. **(i)** On the one hand, $\mathcal{L}_B$(ask,nask,get,tell) $\leqslant \mathcal{L}_{MR}$(ask,nask,get,tell) is immediate by considering the following coder:

$$\begin{aligned} \mathcal{C}(ask(t)) &= (\{+t\},\{\}) \\ \mathcal{C}(get(t)) &= (\{+t\},\{-t\}) \\ \mathcal{C}(nask(t)) &= (\{-t\},\{\}) \\ \mathcal{C}(tell(t)) &= (\{\},\{+t\}) \end{aligned}$$

(ii) On the other hand, $\mathcal{L}_{MR}$(ask,nask,get,tell) $\not\leqslant \mathcal{L}_B$(ask,nask,get,tell) is established by contradiction, using pattern 3 of non embedding by transitivity. Indeed, assuming that $\mathcal{L}_{MR}$(ask,nask,get,tell) $\leqslant \mathcal{L}_B$(ask,nask,get,tell), as $\mathcal{L}_B$(ask,nask,get,tell) $= \mathcal{L}_B$(nask,get,tell), one would have $\mathcal{L}_{MR}$(nask, tell) $\leqslant \mathcal{L}_{MR}$(ask,nask,get,tell) $\leqslant \mathcal{L}_B$(ask,nask,get,tell) $\leqslant \mathcal{L}_B$(nask,get,tell) which has been proved impossible in proposition 26. □

3.3 BachT, MRT and the thesis

Figures 3.5 and 3.6 summarize the expressiveness results of BachT and MRT discussed in the previous section. A more compact version is provided in Figure 3.7 by using a three dimensional perspective. To further stress this perspective dashed arrows have been used instead of plain ones, whereas the semantics (of embedding) behind them is unchanged.

Besides the compact perception, Figure 3.7 allows to present our thesis in a pictorial way. Indeed, by introducing in several ways multiplicity on BachT, we shall actually design languages that fits between the BachT and MRT languages while keeping the same hierarchy between sublanguages, namely by keeping the geometry of the relations depicted in the figure. Figure 3.8 illustrates such an extension on the Dense Bach language which we will introduce in chapter 4.

3.4 Conclusion

In this chapter we have defined two languages BachT and MRT. The first one is a simplified version of a dialect of Linda, developped in Namur. The second one is a Gamma-like language, inspired by a chemical reaction metaphor, with a more global perception of the data. Both languages manipulate tokens in place of more structured tuples in the aim of foccussing on the core coordination features. The four primitives of BachT work with only one token at a time. On the contrary, MRT is able to work atomically with many instances of different tokens. As

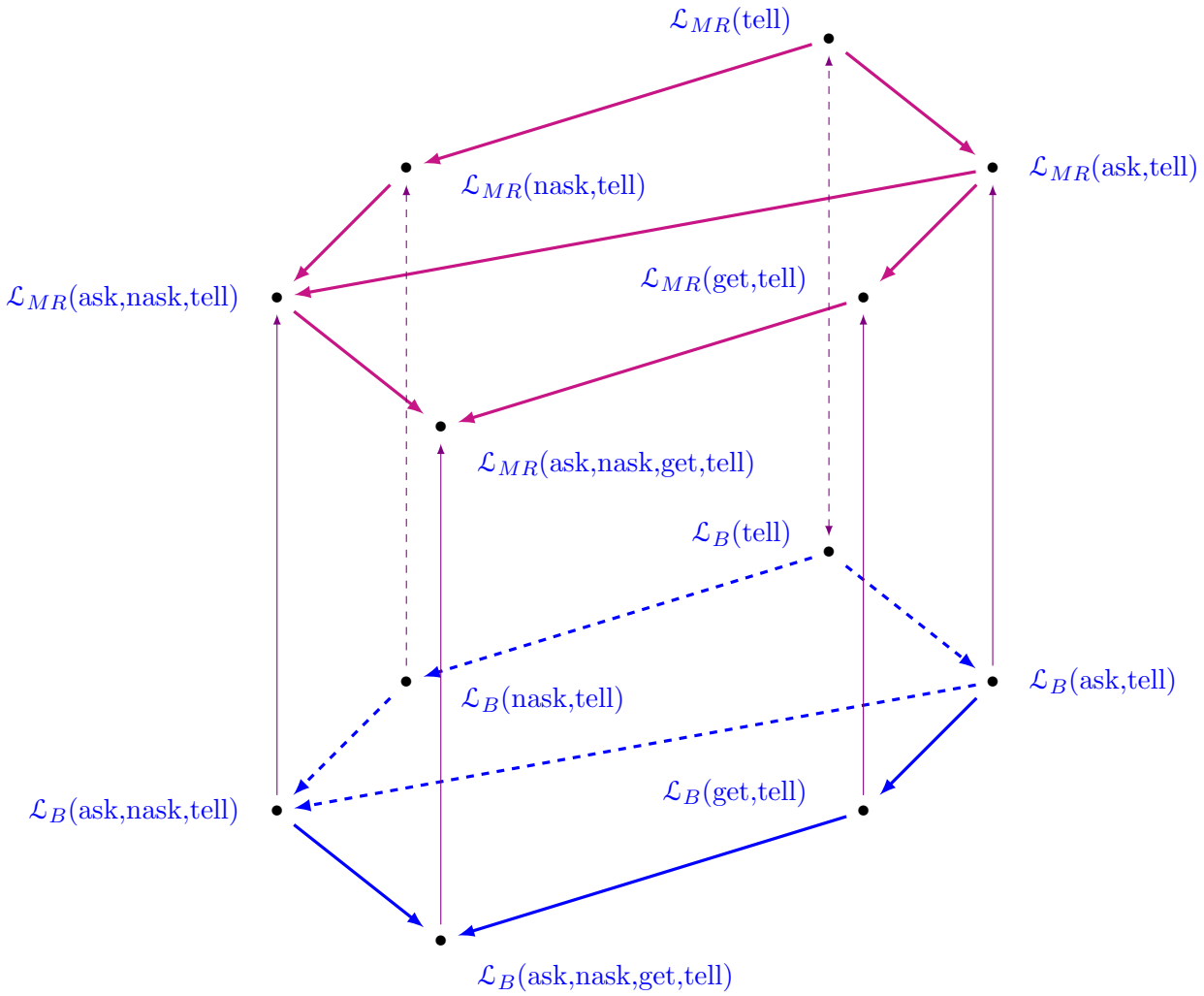


Figure 3.7: Three-dimensional representation of the expressiveness relations between the different sublanguages of BachT and MRT.

will be appreciated in the next chapters, the languages we shall define consist of extensions of BachT, with the capability to manipulate atomically many instances of a same token. In doing so they sit in between the BachT language and the MRT language.

In a second section, we have analyzed the expressiveness of BachT and MRT. In order to do so, we have used the notion of modular embedding introduced in [dBP94]. Following the lines of [BJ03a, BJ98, BJ99], we have first compared the expressive power of sublanguages of BachT with each other. Then we have proceeded with the expressiveness comparison between BachT and MRT. The results of the comparisons depends upon the different considered sublanguages. For the BachT language alone, starting from the sublanguage $\mathcal{L}_B$(tell) one observes an increase in expressiveness, when adding the primitives ask and nask. Nevertheless those

two are not comparable with each other. The introduction of the get primitive is more expressive than the ask primitive when comparing $\mathcal{L}_B$(ask,tell) and $\mathcal{L}_B$(get,tell), but is equivalent to $\mathcal{L}_B$(ask,get,tell). The introduction of the primitive ask in a language where get is already present produces no specific increase of expressiveness. This equivalence is again observed in the comparison of $\mathcal{L}_B$(ask,nask,get,tell) with $\mathcal{L}_B$(nask,get,tell). The introduction of the nask primitive produces a more expressive version of the language. So is it for $\mathcal{L}_B$(ask,nask,tell) with regard to $\mathcal{L}_B$(ask,tell). But there is no comparison possible between $\mathcal{L}_B$(ask,nask,tell) and $\mathcal{L}_B$(get,tell) and its equivalent $\mathcal{L}_B$(ask,get,tell).

The comparison between the BachT language and the MRT language follows the same global schema of expressive relations. However the MRT language is in general more expressive than the BachT language.

Most of the propositions presented in this chapter are already exposed in previous work by J.M. Jacquet and A. Brogi [BJ98, BJ99, BJ03a, BJ03b]. The proofs follow also their reasoning. For BachT the proof of proposition 10 is another way to establish the counter-example. For MRT, proof of proposition 17 comes from [BJ03b]. It has inspired proofs of propositions 22, 23, 24 and 30, that have not been published before. Proof of proposition 18 is inspired from [BJ03b], and is the basis of proofs of propositions 21 and 26 that have also not been published before. Proofs of propositions 19 and 20, as well as those for propositions 27 to 29, and those from 31 to 34 follow similar proof schema inspired from BachT, and have not been published before. Finally, with respect to the work of Jacquet and Brogi, another contribution is to have isolated generic patterns and proof techniques.

For BachT, we have introduced the patterns of sublanguage inclusion, of transitivity and of non embedding by transitivity. The two last ones make use of the transitivity and of previous results on the expressiveness relations between sublanguages in order to establish results more easily. Those three patterns are also valid for the MRT language. However two other patterns specific for this second language have been presented: the pattern of presence and the pattern of absence. They work by contradiction, starting from a coded agent expressed in a normal form. They establish that none of the alternatives in the normal form can exist, leading to an empty form of the coded expression, which is impossible. The second pattern is a special case of the first one. Considering two different elements, it establishes that, on the one hand, their coding share no element, and, on the other hand, that they share at least one element. In both cases this leads again to reduce the coding of a agent to an empty statement, which is absurd.

With respect to [BJ98, BJ99, BJ03a, BJ03b], we have also introduced two proof techniques to highlight several reasonings common to various proofs.

Our goal in writing section 3.2 was to provide the reader with a background on the main reasoning techniques that we shall use later to test the expressiveness of the languages we shall propose. As a result, not all the details have been exposed in this chapter. However, the reader interested by them is referred to appendix A where the proofs are presented extensively.

Nonetheless, as discussed in the third section, the expressiveness results sketched in this chapter allow us to present our thesis pictorially : indeed, in subsequent chapters, we shall introduce extensions of BachT aiming at tackling multiplicity and that fit in between BachT and MRT while preserving the sublanguages hierarchies of BachT and MRT.

A tabulated result of the expressiveness studies between the different sublanguages of BachT and MRT is presented in Table 3.1. All the possible sublanguages are written in line as well as in column of this table. The intersection indicates for every pair their expressiveness relation, as well as a reference to the proof. A number points to the proof developed in the thesis and a reference indicates a publication from other authors.

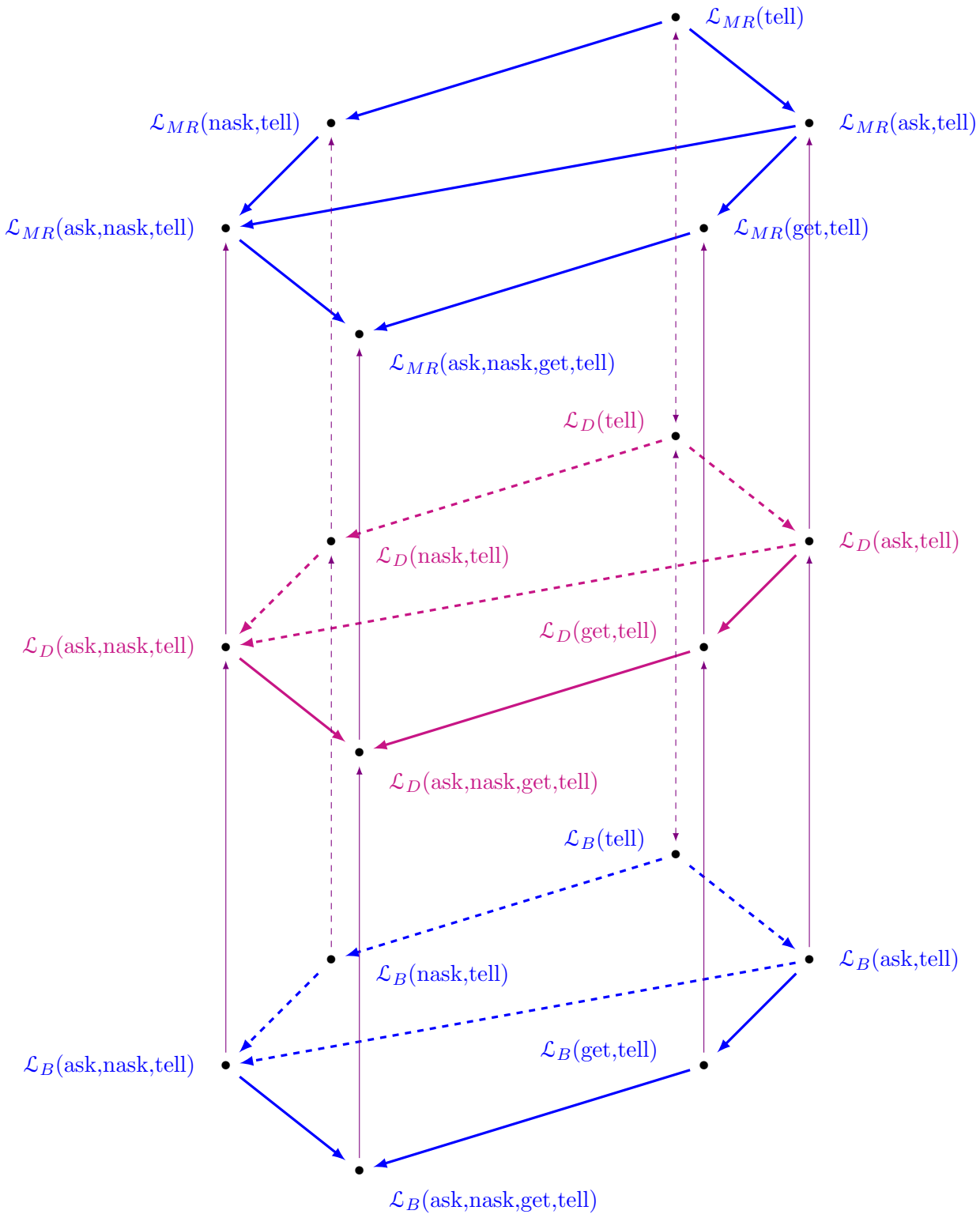


Figure 3.8: Three-dimensional representation of the expressiveness relations between the different sublanguages of BachT, Dense Bach and MRT.

Put reduced figure here	$\mathcal{L}_B$(tell), $\mathcal{L}_{MR}$(tell)	$\mathcal{L}_B$(ask,tell)	$\mathcal{L}_B$(nask,tell)	$\mathcal{L}_B$(get,tell) $\mathcal{L}_B$(ask,get,tell)	$\mathcal{L}_B$(ask,nask,tell)	$\mathcal{L}_B$(nask,get,tell) $\mathcal{L}_B$(ask,nask,get,tell)	$\mathcal{L}_{MR}$(ask,tell)	$\mathcal{L}_{MR}$(nask,tell)	$\mathcal{L}_{MR}$(get,tell) $\mathcal{L}_{MR}$(ask,get,tell)	$\mathcal{L}_{MR}$(ask,nask,tell)	$\mathcal{L}_{MR}$(nask,get,tell) $\mathcal{L}_{MR}$(ask,nask,get,tell)
$\mathcal{L}_B$(tell), $\mathcal{L}_{MR}$(tell)	= 16	$\prec$ 2	$\prec$ 3	$\prec$ 2	$\prec$ 2	$\prec$ 2	$\prec$ 2	$\prec$ 2	$\prec$ 2	$\prec$ 2	$\prec$ 2
$\mathcal{L}_B$(ask,tell)		=	$\wr$ 4	$\prec$ 7	$\prec$ 6	$\prec$ 2	$\prec$ 17	$\wr$ 20	$\prec$ 2	$\prec$ 2	$\prec$ 2
$\mathcal{L}_B$(nask,tell)			=	$\wr$ 8	$\prec$ 5	$\prec$ 2	$\wr$ 19	$\prec$ 18	$\wr$ 31	$\prec$ 2	$\prec$ 2
$\mathcal{L}_B$(get,tell), $\mathcal{L}_B$(ask,get,tell)				= 9	$\wr$ 10	$\prec$ 11	$\wr$ 22	$\wr$ 28	$\prec$ 30	$\wr$ 29	$\prec$ 2
$\mathcal{L}_B$(ask,nask,tell)					=	$\prec$ 13	$\wr$ 24	$\wr$ 21	$\wr$ 33	$\prec$ 25	$\prec$ 2
$\mathcal{L}_B$(nask,get,tell), $\mathcal{L}_B$(ask,nask,get,tell)						= 12	$\wr$ 23	$\wr$ 26	$\wr$ 32	$\wr$ 27	$\prec$ 34
$\mathcal{L}_{MR}$(ask,tell)							=	$\wr$ [BJ03b]	$\prec$ [BJ03b]	$\prec$ [BJ03b]	$\prec$ [BJ03b]
$\mathcal{L}_{MR}$(nask,tell)								=	$\wr$ [BJ03b]	$\prec$ [BJ03b]	$\prec$ [BJ03b]
$\mathcal{L}_{MR}$(get,tell), $\mathcal{L}_{MR}$(ask,get,tell)									=	$\wr$ [BJ03b]	$\prec$ [BJ03b]
$\mathcal{L}_{MR}$(ask,nask,tell)										=	$\prec$ [BJ03b]
$\mathcal{L}_{MR}$(nask,get,tell), $\mathcal{L}_{MR}$(ask,nask,get,tell)											=

Table 3.1: Table summarizing the expressiveness comparisons between the different sublanguages of BachT and MRT.

Part II

Language Design

Chapter 4

The Dense Bach Language

After having introduced the background material of the thesis, we now present our family of languages. In a first one, we complement the BachT language by the deposit and/or retrieval of multiple occurrences of tuples in a shared space. This concretely amounts to enriching the four primitives of BachT with a notion of density. We call this language Dense Bach. It is presented in a similar way to Bach. We first formally define this notion of density as well as the new primitives and grammar of the language. Then, we propose an operational semantics based on a transition system. Examples of possible applications of the language are subsequently described. They tackle such various domains as commerce, security and smart cities. Finally, in a conclusion section, we compare our new language with related work.

4.1 Definition of the language

4.1.1 Language issues

As mentioned before, in the thesis we focus on tuples without structures, called *tokens*. It is however worth noting that by following the developments of [BJL06] the next results can be lifted to more general structures involving enhanced matching techniques.

The following definition formalizes the tokens and the way to attach a density to them.

Definition 10. Let *Stoken* be a enumerable set, the elements of which are subsequently called *tokens* and are typically represented by the letters t and u.

Define the association of a token t and a strictly positive number $n \in \mathbb{N}_0$ as a *dense token*. Such an association is typically denoted as $t(n)$. Define then the set of dense tokens as the set *SDtoken*. Note that since Stoken and $\mathbb{N}$ are both enumerable, the set SDtoken is also enumerable.

Intuitively, a dense token $t(m)$ represents the simultaneous presence of m occurrences of t. As a result, $\{t(m)\}$ is subsequently used to represent the multiset $\{t, \cdots, t\}$ composed of these m occurrences. Moreover, given two multisets of tokens σ and τ, we shall use $\sigma \cup \tau$ to denote the multiset union of elements of σ and τ. As a particular case, by slightly abusing the syntax in writing $\{t(m), t(n)\}$, we have that $\{t(m)\} \cup \{t(n)\} = \{t(m), t(n)\} = \{t(m+n)\}$. Finally, we shall use $\sigma \uplus \{t(m)\}$ to denote, on the one hand, the multiset union of σ and $\{t(m)\}$, and, on the other hand, the fact that t does not belong to σ.

The primitives of the language under consideration extend those of the BachT language to dense tokens. Accordingly, $tell(t(m))$ atomically puts m occurrences of t on the store and $ask(t(m))$ together with $get(t(m))$ require the presence of at least m occurrences of t with the latter removing m of them. Dually, $nask(t(m))$ verifies that there are less than m occurrences of t.

The following definition provides a formal grammar for the extended language.

Definition 11. Define the set of dense token-based primitives $\mathcal{T}_d$ as the set of primitives T_d generated by the following grammar:

$$T_d \quad ::= \quad tell(t(m)) \mid ask(t(m)) \mid get(t(m)) \mid nask(t(m))$$

where t represents a token and m a strictly positive natural number.

With this definition, we are now in a position to define the language we shall consider. The statements of this language, also called *agents*, are defined from the *tell*, *ask*, *get* and *nask* primitives by possibly combining them by the non-deterministic choice operator $+$, the parallel operator, denoted by the $||$ symbol, and the sequential operator, denoted by the $;$ symbol. The formal definition is as follows.

Definition 12. Define the Dense Bach language $\mathcal{L}_{DB}$ similarly to definition 5 of the BachT language but by taking dense token-based primitives T_d :

$$A \quad ::= \quad T_d \mid A \,;\, A \mid A \,||\, A \mid A \;+\; A$$

Subsequently, we shall consider sublanguages formed similarly but by considering only subsets of these primitives. In that case, if $\mathcal{H}$ denotes such a subset, then we shall write the induced sublanguages as $\mathcal{L}_{DB}(\mathcal{H})$. Note that for the latter sublanguages, the *tell*, *ask*, *nask* and *get* primitives are associated with the basic pairs described above.

4.1.2 Transition system

As for BachT, a semantics is defined on the basis of a transition system. Our configuration consists of agents (summarizing the current state of the agents running on the store) and a

$$(\mathbf{T_d}) \quad \frac{m \in \mathbb{N}_0}{\langle\, tell(t(m)) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{t(m)\} \,\rangle}$$

$$(\mathbf{A_d}) \quad \frac{m \in \mathbb{N}_0}{\langle\, ask(t(m)) \mid \sigma \cup \{t(m)\} \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{t(m)\} \,\rangle}$$

$$(\mathbf{G_d}) \quad \frac{m \in \mathbb{N}_0}{\langle\, get(t(m)) \mid \sigma \cup \{t(m)\} \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle}$$

$$(\mathbf{N_d}) \quad \frac{n < m}{\langle\, nask(t(m)) \mid \sigma \uplus \{t(n)\} \,\rangle \longrightarrow \langle\, E \mid \sigma \uplus \{t(n)\} \,\rangle}$$

Figure 4.1: Transition rules for dense token-based primitives (Dense Bach)

multi-set of tokens (denoting the current state of the store). In order to express the termination of the computation of an agent, we also extend as before the set of agents by adding a special terminating symbol E that can be seen as a completely computed agent. For uniformity purposes, we also abuse the language by qualifying E as an agent and to meet the intuition, we shall again simplify agents of the form $(E; A)$, $(E \parallel A)$ and $(A \parallel E)$ as A. This is technically achieved by defining the extended sets of agents as $\mathcal{L}_{DB} \cup \{E\}$ and by justifying the simplifications by imposing a bimonoid structure. Note that no transition rules are therefore used to express these properties.

Figure 4.1 provides the transitions for the dense token-based primitives. Rule (T_d) states that for any store σ and any token t with density m, the effect of the *tell* primitive is to enrich the current set of tokens with m occurrences of token t. Note that $\cup$ denotes multi-set union. Rules (A_d) and (G_d) specify the effect of *ask* and *get* primitives, both requiring the presence of at least m occurrences of t, but the latter also consuming them. Rule (N_d) defines the *nask* primitive, which tests for the absence of m occurrences of t. The rule is successful for any occurences of the token t, provided that they are less than m. It is also worth observing that thanks to the notation $\sigma \uplus \{t(m)\}$ one is sure that t does not occur in σ and consequently that there are exactly n occurrences of t. This does not apply for rules (A_d) and (G_d) for which it is sufficient to assume the presence of at least m occurrences, allowing σ to contain others.

For the sake of completeness, Figure 4.2 recalls the transition rules for the BachT language. As easily observed, they amount to the rules of Figure 4.1 where the density m is taken to be 1, and the union symbol is interpreted on multi-sets.

Finally, Figure 4.3 details the usual rules for sequential composition, parallel composition,

$$(\mathbf{T}) \qquad \langle\, tell(t) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{t\} \,\rangle$$

$$(\mathbf{A}) \quad \langle\, ask(t) \mid \sigma \cup \{t\} \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{t\} \,\rangle$$

$$(\mathbf{G}) \qquad \langle\, get(t) \mid \sigma \cup \{t\} \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle$$

$$(\mathbf{N}) \qquad \frac{t \notin \sigma}{\langle\, nask(t) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle}$$

Figure 4.2: Transition rules for token-based primitives (BachT)

$$(\mathbf{S}) \quad \frac{\langle A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}{\langle A \;;\; B \mid \sigma \rangle \longrightarrow \langle A' \;;\; B \mid \sigma' \rangle}$$

$$(\mathbf{P}) \quad \frac{\langle A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}{\begin{array}{c}\langle A \parallel B \mid \sigma \rangle \longrightarrow \langle A' \parallel B \mid \sigma' \rangle \\ \langle B \parallel A \mid \sigma \rangle \longrightarrow \langle B \parallel A' \mid \sigma' \rangle\end{array}}$$

$$(\mathbf{C}) \quad \frac{\langle A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}{\begin{array}{c}\langle A \;+\; B \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle \\ \langle B \;+\; A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle\end{array}}$$

Figure 4.3: Transition rules for the operators

interpreted in an interleaving fashion, and non-deterministic choice.

4.2 Applications

Several potential applications of Dense Bach have been suggested in chapter 1. As a proof of concept, we now encode them following the syntax of the new language.

4.2.1 Commerce

Different fields of application have been considered, mainly in the commercial, security and public health fields. Concerning the commercial context, let us suppose available an application proposing to evaluate the quality of service offered by a plumber. Let us assume that only positive appreciations are taken into account and that the choice must be made between good or excellent. If a first plumber is identified by a token of the form `plumber_id_1`, then a client could assess it by inserting two times the token on the store if he considers the service as excellent, and only once in case his appreciation is good. Using the primitives of our Dense

Bach language, the process of appreciation for an excellent service is then represented by the following instruction:

$$tell(plumber\_id1(2))$$

whereas in the case of a good one, it is represented by the following instruction:

$$tell(plumber\_id1(1))$$

Supposing the threshold of quality being fixed at 100, a new client consulting the store to obtain information about this plumber will introduce the following request:

$$ask(plumber\_id1(100))$$

Any positive answer would mean that the global appreciation of the plumber is at least of 100. If the application permits to introduce negative appreciations too, a client could express his dissatisfaction by retrieving one token `plumber_id_1` if the service is bad, or two in case of a terrible one. The corresponding instructions are then the following:

$$get(plumber\_id1(1))$$

and

$$get(plumber\_id1(2))$$

This application has the disadvantage to propose a global appreciation, by totalizing the good and excellent appreciations for every plumber. Perhaps a new customer could be interested to obtain more finely-shaded answers. For instance the customer could build his opinion by considering a plumber as relevant if he has at least 20 excellent appreciations, or at least 50 good appreciations. The tokens to be considered on the store must reflect this new point of view, for instance by representing the good appreciation of a plumber with `plumber_id1_good` and with `plumber_id1_excellent` in case of excellence. Every plumber is then represented on the store by two kinds of tokens. On this basis the evaluation of an excellent plumber could be done through the following instruction:

$$tell(plumber\_id1\_excellent(2))$$

A plumber could then be relevant to the positive opinion as previously defined if he satifies the following request:

$$ask(plumber\_id1\_good(50)) \ + \ ask(plumber\_id1\_excellent(20))$$

In this new representation, it is still possible to permit the customer to express its dissatisfaction. Moreover by associating with the plumber a token `plumber_id1_bad` for the bad appreciations and `plumber_id1_terrible` for a terrible one, it becomes also possible to refine the opinion, for instance by asking the plumber to have less than 5 bad appreciations, in addition to the criteria for the positive appreciations. The global request is then the following:

$$nask(plumber\_id1\_bad(5)) \ + \ ask(plumber\_id1\_good(50)) \ + \ ask(plumber\_id1\_excellent(20))$$

Finally, a customer would also like to verify if two plumbers could satisfy to a same set of minimum criteria to be selected, keeping the final choice of selection between those two equivalent workers. For instance if the criteria is to have less than 2 terrible appreciations and more than 30 excellent appreciations, the request is:

$$\begin{array}{c} nask(plumber\_id1\_terrible(2)) \ ; \ ask(plumber\_id1\_excellent(30)) \ || \\ nask(plumber\_id2\_terrible(2)) \ ; \ ask(plumber\_id2\_excellent(30)) \end{array}$$

This example can easily be adapted for the evaluation of a taxi driver, as the one exposed in section 1.1 of chapter 1 of introduction.

4.2.2 Security

In the security field, let us imagine an application that informs a customer of the underground about the motion conditions in its usual metro station during a peak hour. Let us assume that that can be evaluated through the periodic measure of the number of metro users present in the station, represented by a token `number`. For a no-saturation condition fixed to a maximum threshold of 500 people, then the decision of the client to enter the station could be subject to the answer to the following request:

$$nask(number(500))$$

Let us now imagine a user, present in the station, who will leave out quickly this space. This user is interested to know the entering flux of travellers for every exit . If this is represented

for a specific entry by the token `entry_id`, the best exit to be chosen is the one satisfying the following request, for a fixed threshold of easiness of 50 entering people, and for three available exits:

$$nask(entry\_id1\_(50)) \ + \ nask(entry\_id2\_(50)) \ + \ nask(entry\_id3\_(50))$$

Let us imagine that a security guard wants to be informed of the global safety conditions in the station, these conditions being measured from the incoming and outgoing flux `in_flux` and `out_flux` of users through the entry doors. If a door is considered to be safe when both flux are less than 30, then for two doors and with the colour codes `green` for a safe condition, `orange` for a light heavy condition and `red` for an overloaded condition, the global situation can be evaluated with the following request:

$$\begin{array}{rc}
(nask(door1_in_flux(30)) \ ; & (nask(door1_out_flux(30)) \ ; tell(green) \\
 & +ask(door1_out_flux(30)) \ ; tell(orange))) \\
+(ask(door1_in_flux(30)) \ ; & (nask(door1_out_flux(30)) \ ; tell(orange) \\
 & +ask(door1_out_flux(30)) \ ; tell(red))) \\
 & || \\
(nask(door2_in_flux(30)) \ ; & (nask(door2_out_flux(30)) \ ; tell(green) \\
 & +ask(door2_out_flux(30)) \ ; tell(orange))) \\
+(ask(door2_in_flux(30)) \ ; & (nask(door2_out_flux(30)) \ ; tell(orange) \\
 & +ask(door2_out_flux(30)) \ ; tell(red)))
\end{array}$$

4.2.3 Smart cities

In the field of the management of smart cities, let us imagine an application that can divert the flux of vehicles following their type and their number. Let us suppose that every speedway entering a city is equipped with a gate that can register and count every type of vehicle. With the tokens `motorbike`, `car` and `lorry`, every positive counting is obtained by the following instruction:

$$tell(motorbike(1)) \ + \ tell(car(1)) \ + \ tell(lorry(1))$$

If the number of counted vehicles becomes higher than a certain threshold, i.e. 50, the following instruction can divert the flux in some specific direction, for every type of vehicle:

$$ask(motorbike(50)) \,;\, tell(turn\_right) \;+\; ask(car(50)) \,;\, tell(straight\_on) \;+\; ask(lorry(50)) \,;\, tell(turn\_left)$$

with the tokens `turn_right`, `straight_on` and `turn_left` indicating the respective directions.

4.3 Conclusion

In this chapter we have presented an extension of the Linda-like language Bach, aiming at introducing a notion of density to the tuples. The intuition behind is that, on the one hand, the more a tuple appears on a tuple space, the more it is of interest. On the other hand, tuples are of interest for the test of their presence or absence only if they appear in a sufficient number of occurences. The new resulting language has been named Dense Bach. Examples have been presented to describe the possible uses of the language in concrete situations.

In the literature, several pieces of work have tackled similar issues : [BKZ99, BGZ97, BGZ, Zav98a, Zav98b]. However, they do not deal with the notion of density that we have introduced.

The articles [BGLZ04, BGLG05] introduce the multiplicity of tokens by decorating them with an extra field in order to investigate how probabilities and priorities can be introduced in the Linda coordination model. The notion of density resembles but is not identical to the association of weights to tuple. Indeed density does not modify the tokens on the store, as it does not modify the matching function so as to retrieve the token with the highest weight. In contrast, it modifies the tokens primitives so as to be able to atomically put several occurrences of a tuple on the store and check for the presence or absence of a number of occurrences. This facility of handling atomically several occurrences produces a real increase in expressiveness, as it will be established in chapter 6.

Similarly to the weight of [BGLZ04, BGLG05] and our notion of density, Viroli and Casadei [VC09] propose a stochastic extension of the Linda framework, with a notion of tuple concentration. The syntax of the token space is modeled by means of a calculus, with an operational semantics given as an hybrid CTMC/DTMC model. It describes the behavior of tell, ask and get like primitives, but no equivalent of nask.

Chapter 5

Dense Bach with Distributed Density

In an attempt to provide Dense Bach with the same property of MRT to handle many tokens at once, a natural extension consists in replacing in the primitives a token by a list of tokens, each with its own density. For instance, the primitive $ask(t(1), u(2), v(3))$ would succeed on a store containing one occurrence of t, two of u and three of v. Dually, the computation of $tell(t(1), u(2), v(3))$ would result in adding one occurrence of t on the store, two of u and three of v.

The first section of this chapter formally defines this extension of Dense Bach, subsequently named Vectorized Dense Bach or more simply VD-Bach. It additionnally specifies the corresponding transition rules and provides an example of application aiming at modeling the problem of migrants in direct drive with the current events.

It turns out that VD-Bach can be used to encode more powerful versions distributing a density on a list of tokens possibly under cardinality constraints. This will be particularly useful in the coding of the migrant problem. The second section of this chapter tackles this new extension. Finally the third section concludes and introduces the following chapters, concerned with the comparison of the expressiveness study of the languages.

5.1 Definition of VD-Bach

5.1.1 Language issues

As underlined in the introduction, a natural extension to the dense tokens introduced in definition 10 consists in considering sets of such tokens. To avoid using unnecessary brackets, we shall

slightly abuse notations and use lists of dense tokens, which we shall subsequently designate as vectors of dense tokens. The intuition remains however that of sets, with the order of the dense tokens being meaningless.

Definition 13. Define a *vector of dense tokens* as a list $t_1(m_1), \cdots, t_n(m_n)$ of dense tokens. Such a vector is subsequently denoted as $\overrightarrow{t(m)}$. Define $SVDtoken$ as the set of vectors of dense tokens.

With this definition in mind, a natural extension of Dense Bach consists in lifting the arguments of the primitives to vectors of dense tokens, with the intuition that what was required in chapter 4 has now to be required for all the elements of the vectors. We are thus led to the following definitions.

Definition 14. Define the set of vectorized dense token-based primitives $\mathfrak{T}_v$ as the set of primitives T_v generated by the following grammar:

$$T_v \quad ::= \quad tell(\overrightarrow{t(m)}) \mid ask(\overrightarrow{t(m)}) \mid get(\overrightarrow{t(m)}) \mid nask(\overrightarrow{t(m)})$$

where $\overrightarrow{t(m)}$ represents a vector of dense tokens.

Definition 15. Define the Vectorized Dense Bach language similarly to definition 5 of the BachT language but by taking vector of dense token-based primitives T_v :

$$A \quad ::= \quad T_v \mid A \,;\, A \mid A \,||\, A \mid A \;+\; A$$

Subsequently, we shall consider sublanguages formed similarly but by considering only subsets of these primitives. In that case, if $\mathcal{H}$ denotes such a subset, then we shall write the induced sublanguages as $\mathcal{L}_V(\mathcal{H})$. Moreover we shall abuse language and also note $\mathcal{L}_V$ for the super language containining all the primitives.

5.1.2 Transition system

As for Dense Bach, a semantics is defined on the basis of a transition system. Our configuration consists again of agents (summarizing the current state of the agents running on the store) and a multi-set of tokens (denoting the current state of the store). In order to express the termination of the computation of an agent, we also extend as before the set of agents by adding a special terminating symbol E that can be seen as a completely computed agent. For uniformity purposes, we also abuse the language by qualifying E as an agent and to meet the intuition, we shall again simplify agents of the form $(E; A)$, $(E \,||\, A)$ and $(A \,||\, E)$ as A. This is technically achieved by defining the extended sets of agents as $\mathcal{L}_V \cup \{E\}$ and by justifying the simplifications by imposing a bimonoid structure. Note that no transition rules are therefore used to express these properties.

$$(\mathbf{T_v}) \qquad \frac{m_1, \cdots, m_n \in \mathbb{N}_0}{\langle\, tell(t_1(m_1), \cdots, t_n(m_n)) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{t_1(m_1), \cdots, t_n(m_n)\} \,\rangle}$$

$$(\mathbf{A_v}) \qquad \frac{m_1, \cdots, m_n \in \mathbb{N}_0}{\langle\, ask(t_1(m_1), \cdots, t_n(m_n)) \mid \sigma \cup \{t_1(m_1), \cdots, t_n(m_n)\} \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{t_1(m_1), \cdots, t_n(m_n)\} \,\rangle}$$

$$(\mathbf{G_v}) \qquad \frac{m_1, \cdots, m_n \in \mathbb{N}_0}{\langle\, get(t_1(m_1), \cdots, t_n(m_n)) \mid \sigma \cup \{t_1(m_1), \cdots, t_n(m_n)\} \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle}$$

$$(\mathbf{N_v}) \qquad \frac{m_1, \cdots, m_n \in \mathbb{N}_0, p_1 < m_1, \cdots, p_n < m_n}{\langle\, nask(t_1(m_1), \cdots, t_n(m_n)) \mid \sigma \uplus \{t_1(p_1), \cdots, t_n(p_n)\} \,\rangle \longrightarrow \langle\, E \mid \sigma \uplus \{t_1(p_1), \cdots, t_n(p_n)\} \,\rangle}$$

Figure 5.1: Transition rules for vectorized dense token-based primitives (VD-Bach)

Figure 5.1 provides the transitions for the vectorized dense token-based primitives. For readability purposes, in contrast to definition 14, we have written the arguments of primitives as explicit vectors of dense tokens, in the form of $t_1(m_1), \cdots, t_n(m_n)$.

Rule (T_v), (A_v) and (G_v) generalize the corresponding rules (T_d), (A_d) and (G_d) of chapter 4 from dense tokens to vectors of dense tokens. As a result, rule (T_v) asserts that telling a vector of dense tokens amounts to adding each of them with the corresponding density on the store. Similarly, rule (A_v) requires for an ask primitive to succeed the presence, for each token t_i, of at least m_i occurrences on the store. According to rule (G_v) the behavior of a get primitive performs such a test for presence but also removes m_i occurrences of t_i on the store. Finally, rule (N_v) requires, for each token t_i, the absence of m_i occurrences. It is here worth noting that, in contrast to BachT and Dense Bach, the behavior of the nask primitive is not the negation of that of the ask primitive. Indeed, this interpretation would have required for the nask primitive that, for some token t_i, less than m_i occurrences are present on the store. It will however be handful to have such a nask primitive. The next subsection introduces it and shows how it can be simulated by ask and nask primitives, just introduced.

For the sake of completeness, Figure 5.2 reminds the reader with the usual rules for sequential composition, parallel composition, interpreted in an interleaving fashion, and non-deterministic choice.

5.1.3 Weak negative ask

As just mentioned, rule (N_v) does not give to the nask primitive a dual behavior of the ask primitive. To that end, a new nask primitive, subsequently called weak nask primitive, may be

$$(\mathbf{S}) \quad \frac{\langle A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}{\langle A \ ; \ B \mid \sigma \rangle \longrightarrow \langle A' \ ; \ B \mid \sigma' \rangle}$$

$$(\mathbf{P}) \quad \frac{\langle A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}{\begin{array}{c}\langle A \parallel B \mid \sigma \rangle \longrightarrow \langle A' \parallel B \mid \sigma' \rangle \\ \langle B \parallel A \mid \sigma \rangle \longrightarrow \langle B \parallel A' \mid \sigma' \rangle\end{array}}$$

$$(\mathbf{C}) \quad \frac{\langle A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}{\begin{array}{c}\langle A \ + \ B \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle \\ \langle B \ + \ A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle\end{array}}$$

Figure 5.2: Transition rules for the operators

$$(\mathbf{W_v}) \quad \frac{m_1, \cdots, m_n \in \mathbb{N}_0, \{t_1(p_1), \cdots, t_n(p_n)\} \nsubseteq \sigma}{\langle\ wnask(t_1(m_1), \cdots, t_n(m_n)) \mid \sigma)\ \rangle \longrightarrow \langle\ E \mid \sigma\ \rangle}$$

Figure 5.3: Transition rule for the weak nask

introduced. It is referred to as the *wnask* primitive, takes a vector of dense tokens as argument and is defined by rule (W_v) of figure 5.3.

Note that with such a definition, $wnask(\overrightarrow{t(m)})$ succeeds whenever $nask(\overrightarrow{t(m)})$ succeeds. However, the converse is not true. Consider, for instance, the store composed of 2 occurrences of t and 4 of u. In that context, $nask(t(1), u(5))$ does not succeed since, although $4 < 5$ the inequality $2 < 1$ does not hold. However, $wnask(t(1), u(5))$ succeeds since, as multisets, $\{t(2), u(4)\} \nsubseteq \{t(1), u(5)\}$. Rephrased using the notation of rule (N_v), it it required for *nask* that

$$p_1 < m_1 \wedge \cdots \wedge p_n < m_n$$

whereas *wnask* only requires that

$$p_1 < m_1 \vee \cdots \vee p_n < m_n$$

In view of that, it is easy to verify that $wnask(t_1(m_1), \cdots, t_n(m_n))$ can be encoded as follows :

$$nask(t_1(m_1)) + \cdots + nask(t_n(m_n))$$

As a result, *wnask* does not bring an increase of expressiveness.

5.1.4 Application

In order to illustrate a use of this newly defined language, let us imagine the example of an online shopping system. Let us imagine a sporting goods store that is present in five different cities in Belgium : Brussels, Namur, Nivelles, Antwerp and Louvain. All these shops propose the same articles. In order to manage efficiently the number of orders that arrive through the online system, these are distributed on the different shops present in the five cities. Assume that a group of 50 orders arrive and has to be distributed equally between the different shops. This can be achieved through the execution of the following tell primitive :

```
tell(Brussels(10), Namur(10), Nivelles(10), Antwerp(10), Louvain(10)).
```

Assume now that the following maxima of orders to be processed have been imposed for the shops : 200 orders for Brussels, 75 for Namur, 50 for Nivelles, 150 for Antwerp and 70 for Louvain. A check whether these maxima have not been reached can be simulated by executing the following nask primitive :

```
nask(Brussels(200), Namur(75), Nivelles(50), Antwerp(150), Louvain(70)).
```

5.2 On Distributed Density

5.2.1 Definition of a distributed density

In the online shopping problem, we have explicitly distributed the arrival of 50 orders on the shops. A natural extension is to let the execution of the primitive non-deterministically choose the distribution. We are then lead to consider a list of tokens together with a density and to distribute it on the tokens. The following definition formalizes such an association.

Definition 16. Let *Snlt* denote the set of non-empty lists of tokens in which, for simplicity purposes, each token differs from the others. Such a list is typically denoted as $L = [t_1, \ldots, t_p]$ and is thus such that $t_i \neq t_j$ for $i \neq j$. Define a dense list of tokens as a list of *Snlt* associated with a strictly positive integer. Such a dense list is typically represented as $L(m)$, with L the list of tokens and m an integer.

Definition 17. Define the association of a token and a positive integer of $\mathbb{N}$ as an extended dense token.

Note that in contrast to definition 10, we allow here for the association of a token with 0.

The distribution of the density over a list of tokens is formalized through the following distribution function.

Definition 18. Define the distribution of tokens from dense lists of tokens to sets of tuples of extended dense tokens as follows:

$$\mathcal{D}i([t_1, \cdots, t_p](m)) = \{(t_1(m_1), \cdots, t_p(m_p)) : m_1 + \cdots + m_p = m\}$$

Note that, thanks to the definition of extended dense tokens, we assume above that the m_i's are positive integers. For the sake of simplicity, we shall call the set $\mathcal{D}i([t_1, \cdots, t_p](m))$ the distribution of m over $[t_1, \cdots, t_p]$.

The distribution of an integer m over a list of tokens L has the potential to express the behavior of the BachT primitives extended with dense lists of tokens as arguments. Indeed, telling a dense list amounts to telling atomically the $t_i[m_i]$'s of a tuple defined above. Asking or getting a dense list requires to check that a tuple of $\mathcal{D}i([t_1, \cdots, t_p](m))$ is present on the considered store. For the negative ask, the requirement is that none of the tuple is present. For the ease of writing and to make this latter concept clear, we introduce the following concept of intersection.

Definition 19. Let m be a positive integer, $L = [t_1, \cdots, t_p]$ be a list of tokens and σ a store. We define $\mathcal{D}i(L(m)) \sqcap \sigma$ as the following set of tuples of dense tokens :

$$\mathcal{D}i(L(m)) \sqcap \sigma = \{(t_1(m_1), \ldots, t_p(m_p)) \in \mathcal{D}i(L(m)) : \{t_i(m_i)\} \subseteq \sigma\}$$

From an implementation point of view, it is worth observing that one may give a syntactical characterization of the emptyness of such an intersection.

Definition 20. Given a store σ and a dense list $L(m)$, with $L = [t_1, \cdots, t_p]$, we denote by $Max(\sigma, L(m))$ the tuple $(t_1(m_1), \ldots, t_p(m_p))$ where the m_i's denote the number of occurrences of t_i in σ. Moreover, we denote by $SMax(\sigma, L(m))$ the sum $m_1 + \cdots + m_p$

It is easy to establish the following proposition.

Theorem 1. For any dense list of tokens $L(m)$ and any store σ, one has $\mathcal{D}i(L(m)) \sqcap \sigma = \varnothing$ iff $SMax(\sigma, L(m)) < m$.

Proof. Simple verification. □

$$(\mathbf{T_{dbd}}) \quad \frac{(t_1(m_1), \cdots, t_p(m_p)) \in \mathcal{D}i(L(m))}{\langle\, tell(L(m)) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{t_1(m_1), \cdots, t_p(m_p)\} \,\rangle}$$

$$(\mathbf{A_{dbd}}) \quad \frac{\mathcal{D}i(L(m)) \sqcap \sigma \neq \varnothing}{\langle\, ask(L(m)) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle}$$

$$(\mathbf{G_{dbd}}) \quad \frac{(t_1(m_1), \cdots, t_p(m_p)) \in \mathcal{D}i(L(m))}{\langle\, get(L(m)) \mid \sigma \cup \{t_1(m_1), \cdots, t_p(m_p)\} \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle}$$

$$(\mathbf{N_{dbd}}) \quad \frac{m > 0 \text{ and } \mathcal{D}i(L(m)) \sqcap \sigma = \varnothing}{\langle\, nask(L(m)) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle}$$

Figure 5.4: Transition rules for list of token-based primitives (Dense Bach with distributed Density)

5.2.2 Definition of DBD-Bach

We are now in a position to specify the language extension handling dense lists of tokens.

Definition 21. Define the set of dense lists primitives $\mathcal{T}_{dbd}$ as the set of primitives T_{dbd} generated by the following grammar:

$$T_{dbd} \quad ::= \quad tell(L(m)) \mid ask(L(m)) \mid get(L(m)) \mid nask(L(m))$$

where $L(m)$ represents a dense list of tokens.

The transition steps for these primitives are defined in figure 5.4. As suggested above, rule (T_{dbd}) specifies that telling a dense list $L(m)$ of tokens amounts to atomically adding the multiple occurrences $t_i(m_i)$'s of the tokens of a tuple of the distribution of m over L. Note that the selected tuple is chosen non-deterministically, which gives to a tell primitive a non-deterministic behavior as opposed to the tell primitives of BachT and Dense Bach. Rule (A_{dbd}) states that asking for the dense list $L(m)$ amounts to testing that a tuple of the distribution of m over L is in the store, which is technically stated through the non-emptyness of the intersection of the distribution and the store. Rule (G_{dbd}) requires that the tokens of the tuples are removed in the considered multiplicity. Finally, rule (N_{dbd}) specifies that negatively asking $L(m)$ succeeds if m is strictly positive and no tuple of the distribution of m over L is present on the current store.

We are now in a position to define the language Dense Bach with a distribution of the density over a list of tokens. The statements of this language, also called *agents*, are defined

$$\textbf{(S)}\quad \frac{\langle A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\langle A\ ;\ B \mid \sigma\rangle \longrightarrow \langle A'\ ;\ B \mid \sigma'\rangle}$$

$$\textbf{(P)}\quad \frac{\langle A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\begin{array}{c}\langle A \parallel B \mid \sigma\rangle \longrightarrow \langle A' \parallel B \mid \sigma'\rangle \\ \langle B \parallel A \mid \sigma\rangle \longrightarrow \langle B \parallel A' \mid \sigma'\rangle\end{array}}$$

$$\textbf{(C)}\quad \frac{\langle A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\begin{array}{c}\langle A\ +\ B \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle \\ \langle B\ +\ A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle\end{array}}$$

Figure 5.5: Transition rules for the operators

from the tell, ask, get and nask primitives by possibly combining them by the classical choice operator +, used among others in CCS, parallel operator (denoted by the $\parallel$ symbol) and the sequential operator (denoted by the ; symbol). The formal definition is as follows.

Definition 22. Similarly to the definitions 5 and 12, define the Dense Bach with distributed Density language or DBD-Bach for short by taking lists of token-based primitives T_{dbd}:

$$A \quad ::= \quad T_{dbd} \mid A\ ;\ A \mid A \parallel A \mid A\ +\ A$$

Subsequently, we shall consider sublanguages formed similarly but by considering only subsets of these primitives. In that case, if $\mathcal{H}$ denotes such a subset, then we shall write the induced sublanguages as $\mathcal{L}_{DBD}(\mathcal{H})$. Moreover we shall abuse language and also note $\mathcal{L}_{DBD}$ for the super language containining all the primitives.

To study the expressiveness of the languages, a semantics needs to be defined. As done in the previous chapters, we shall use an operational one, based on transition systems and for which the configuration consists of agents (summarizing the current state of the agents running on the store) and a multi-set of tokens (denoting the current state of the store). In order to express the termination of the computation of an agent, we also extend the set of agents by adding a special terminating symbol E that can be seen as a completely computed agent. For uniformity purpose, we abuse the language by qualifying E as an agent. To meet the intuition, we shall always rewrite agents of the form $(E; A)$, $(E \parallel A)$ and $(A \parallel E)$ as A. This is technically achieved by justifying the simplifications by imposing a bimonoid structure.

The rules for the primitives of the languages have been given in Figures 5.1 to 5.4. Figure 5.5 details the usual rules for sequential composition, parallel composition, interpreted in an interleaving fashion, and non-deterministic choice.

5.2.3 Application

To illustrate the DBD-Bach language, let us consider again the online shopping problem and handle the non-deterministic distribution of the orders between the different shops. Assume a group of m orders is to be split into different shops. This can be modeled by the primitive $tell(L(m))$ with `L = [Brussels,Namur,Nivelles,Antwerp,Louvain]` being the list of shops. Indeed, remember that this amounts to identify m_1, m_2, m_3, m_4 and m_5 such that $m_1 + m_2 + m_3 + m_4 + m_5 = m$ and consequently to tell m_1 orders in Brussels, m_2 in Namur, m_3 in Nivelles, m_4 in Antwerp and m_5 in Louvain.

Let us suppose that the orders are grouped in three different services by the online system. Every group of selected `n` orders for processing may be constitued by a non-deterministic selection inside the different services, with the primitive $get(L(n))$. To that end, it is here worth remembering that the execution of this primitive amounts to identifying n_1, n_2 and n_3 such that $n_1 + n_2 + n_3 = n$ and consequently to get n_1 orders in the first service, n_2 orders in the second service and n_3 orders in third service.

Let us finally suppose that Brussels has dispatched its own orders between five internal departments in the shop and that it wants to know whether the number of incoming orders does not exceed the total capacity fixed to 200. The resquest to answer such a question may be modelled by the following primitive:

$$nask([(Brussels\_dept_1, \cdots, Brussels\_dept_5](200)).$$

The general direction could ask the same question to measure the situation in some shops, taking into account their respective maximum capacity. Consider, for instance, Nivelles and Antwerp and assume that $L\_Nivelles$ and $L\_Antwerp$ are their respective lists of departments and that the maximum capacities are respectively 50 and 150 orders. Then the question may be modeled by the following agent :

$$nask(L\_Nivelles(50)) \;||\; nask(L\_Antwerp(150)).$$

5.2.4 Cardinality on tokens

Coding the online shopping problem suggests the further extension of requiring a shop to fulfill a minimal processing of orders while guaranteeing a maximal processing. This can be coded by slightly extending the tokens of a dense list with minimal and maximal numbers. The formal definition is as follows.

Definition 23. Define the association of a token and two positive integers of $\mathbb{N}$ as a *capacity dense token*. Such a token is typically denoted as $t(m, n)$ where t is the token and m, n are the integers.

Definition 24. Let *Snlct* denote the set of non-empty lists of capacity dense tokens in which, for simplicity purposes, each token differs from the others. Such a list is typically denoted as $L = [t_1(m_1, n_1), \ldots, t_p(m_p, n_p)]$ and is thus such that $t_i \neq t_j$ for $i \neq j$. Define a dense list of capacity dense tokens as a list of *Snlct* associated with a strictly positive integer. Such a list is typically represented as $L(m)$, with L the list of capacity dense tokens and m an integer.

The expected extended language is simply obtained by slightly modifying the notion of distribution introduced in definition 18.

Definition 25. Define the cardinality based distribution of tokens from dense lists of capacity tokens to sets of tuples of extended dense tokens as follows:

$$\mathcal{D}c([t_1(m_1, n_1), \cdots, t_p(m_p, n_p)](q)) = \{(t_1(q_1), \cdots, t_p(q_p)) : q_1 + \cdots + q_p = q$$
$$\text{and } m_i \leqslant q_i \leqslant n_i \text{ for any } i \in \{1, \cdots, p\}\}$$

Note that nothing guarantees that the above set is non empty. We shall subsequently called *coherent* those dense lists of capacity based tokens such that their cardinality based distribution is non empty and restrict ourselves to such coherent dense lists in the following.

It is now easy to extend the primitives of the DBD-Bach language.

Definition 26. Define the set of capacity based list primitives $\mathcal{T}_c$ as the set of primitives T_c generated by the following grammar:

$$T_c \quad ::= \quad tell(L(m)) \mid ask(L(m)) \mid get(L(m)) \mid nask(L(m))$$

where $L(m)$ represents a dense list of capacity based tokens.

Definition 27. Define the Capacity based Dense Bach as the language of agents engendered by the following grammar where T_c denotes a capacity based list primitive :

$$A \quad ::= \quad T_c \mid A \,;\, A \mid A \,||\, A \mid A \;+\; A$$

The transition rules derive directly from those of figure 5.4 by replacing the set $\mathcal{D}i(L(m))$ by the corresponding set $\mathcal{D}c(L(m))$. They are listed in figure 5.6 for the sake of completion.

Equipped with the Capacity based Dense Bach language, it is quite easy to model that upon the arrival of m orders the shops take their possibility without going further than some limits. This is indeed coded by a primitive of the form :

$$tell([Brussels(100, 200), Namur(30, 75), Nivelles(25, 50), Antwerp(45, 150), Louvain(25, 70)](m))$$

$$(\mathbf{T_c}) \quad \frac{(t_1(m_1), \cdots, t_p(m_p)) \in \mathcal{D}c(L(m))}{\langle\, tell(L(m)) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{t_1(m_1), \cdots, t_p(m_p)\} \,\rangle}$$

$$(\mathbf{A_c}) \quad \frac{\mathcal{D}c(L(m)) \sqcap \sigma \neq \varnothing}{\langle\, ask(L(m)) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle}$$

$$(\mathbf{G_c}) \quad \frac{(t_1(m_1), \cdots, t_p(m_p)) \in \mathcal{D}c(L(m))}{\langle\, get(L(m)) \mid \sigma \cup \{t_1(m_1), \cdots, t_p(m_p)\} \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle}$$

$$(\mathbf{N_c}) \quad \frac{m > 0 \text{ and } \mathcal{D}c(L(m)) \sqcap \sigma = \varnothing}{\langle\, nask(L(m)) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle}$$

Figure 5.6: Transition rules for capacity based primitives

5.2.5 Translation in VD-Bach

It is quite easy to translate the positive version of DBD-Bach and CD-Bach in terms of VD-Bach.

Indeed, as easily observed, one can code the tell, ask and get primitives of DBD-Bach as follows :

$$\begin{aligned} tell(L(m)) &= \sum_{\vec{v} \in \mathcal{D}i(L(m))} tell(\vec{v}^{\,r}) \\ ask(L(m)) &= \sum_{\vec{v} \in \mathcal{D}i(L(m))} ask(\vec{v}^{\,r}) \\ get(L(m)) &= \sum_{\vec{v} \in \mathcal{D}i(L(m))} get(\vec{v}^{\,r}) \end{aligned}$$

where $\vec{v}^{\,r}$ denotes the vector $\vec{v}$ restricted to its strictly positive dense tokens.

Translating the nask primitive is slightly more complicated in requiring the parallel composition of the weak form of nask of vectors:

$$nask(L(m)) = ||_{\vec{v} \in \mathcal{D}i(L(m))} wnask(\vec{v}^{\,r})$$

Translating the CD-Bach language proceeds similarly by using $\mathcal{D}c(L(m))$ instead of $\mathcal{D}i(L(m))$.

5.3 Conclusion

In this chapter, we have defined an extension of the language Dense Bach, with the property of distributing density on a finite list of tokens. The idea of this extension is to provide the

language with a feature similar to a multi-set language, regarding the capacity to atomically manipulate not only many instances of a same token, as Dense Bach does, but also different tokens.

It is worth stressing that the idea of distributing the density on a vector of tokens is new and, to the best of our knowledge, has never been the object of any proposal in the literature. Combined with the notion of minimum and maximum, it proposes an elegant way to model real-life problems, as our example about the migrants.

The next step is to study the expressiveness of the languages we have proposed. Obviously, in view of the translations we have provided in this chapter, two languages have mainly to be compared : Dense Bach and Vectorized Dense Bach. We shall do so in the next two chapters by comparing these two languages with the BachT and MRT languages presented in chapter 3.

Chapter 6

Expressiveness Study of Dense Bach

This chapter positions Dense Bach from an expressiveness point of view. The first section compares it with BachT. The second section compares it with MRT.

In both cases, we shall use the modular embedding technique proposed by De Boer and Palamidessi (see [dBP94]), in the form slightly redefined in section 3.2.1, and already employed in sections 3.2.4 and 3.2.5.

In both cases, we shall also restrict to the relevant sublanguages, namely those that embody the tell primitive.

6.1 Comparison with BachT

6.1.1 Generic patterns and results

It is first worth observing that the generic patterns introduced in section 3.2.2 also apply in the context of Dense Bach. As a reminder, they embody the following reasonings. The first pattern, named the pattern of sublanguage inclusion, establishes that any language embeds its sublanguages. The second pattern, named pattern of transitivity, takes advantage of transition to establish that $\mathcal{L}_1 \leqslant \mathcal{L}_3$ from the facts that $\mathcal{L}_1 \leqslant \mathcal{L}_2$ and $\mathcal{L}_2 \leqslant \mathcal{L}_3$. The third pattern relies on the contraposition of the second pattern, to establish the non existence of the embedding of $\mathcal{L}_2$ in $\mathcal{L}_3$, namely $\mathcal{L}_2 \not\leqslant \mathcal{L}_3$, from the facts that $\mathcal{L}_1 \leqslant \mathcal{L}_2$ and $\mathcal{L}_1 \not\leqslant \mathcal{L}_3$.

For what concerns the Dense Bach language alone, it is clear by the first pattern (pattern 1 of sublanguage inclusion) that a number of modular embeddings are directly established. This first property is formally expressed into the following proposition.

Proposition 35. $\mathcal{L}_{DB}(\ \psi) \leqslant \mathcal{L}_{DB}(\ \chi)$, for any subsets of ψ, χ of primitives such that $\psi \subseteq \chi$.

As a second result, it is also clear that a simple relation exists between the BachT primitives and their equivalent form in Dense Bach, simply by taking single occurrences, namely a density of 1. As a result, BachT sublanguages are embedded in the corresponding Dense Bach sublanguages, as expressed into the following proposition.

Proposition 36. $\mathcal{L}_{B}(\ \chi) \leqslant \mathcal{L}_{DB}(\ \chi)$, for any subset of χ of primitives.

Proof. The proof is immediate by defining the coder as follows:

$$\begin{array}{rclcrcl} \mathcal{C}(tell(t)) & = & tell(t(1)) & & \mathcal{C}(get(t)) & = & get(t(1)) \\ \mathcal{C}(ask(t)) & = & ask(t(1)) & & \mathcal{C}(nask(t)) & = & nask(t(1)) \end{array}$$

□

More deeply this expresses that BachT is a special case of Dense Bach. As the introduction of density cannot alter the very nature of the four primitives *tell*, *ask*, *nask* and *get*, a similarity between the respective hierarchies of Dense Bach sublanguages and of Bach sublanguages, is to be expected, as showned in Figure 3.5 of Section 3.2.2. Nevertheless, even if the global expressive behaviour stays the same, it is also expected that the complete picture of the two hierarchies will reflect an increase in expressiveness in favour of Dense Bach with regard to BachT. Indeed, apart for the *tell* primitive, the introduction of density gives to *ask*, *nask* and *get* a more efficient atomic behaviour than for their BachT corresponding primitives. As an example the *ask(t(m))* primitive could not be simulated by m times an *ask(t)* primitive, as this one could select m times the same token t, while *ask(t(m))* requires at least m instances of t. The same reasoning is valid for the *get(t(m))* primitive. Concerning the *nask(t(m))* primitive, a succesful result could be obtained even in case of the presence of some tokens t, since they are present in a number smaller than m.

After those general and intuitive results, the next subsections present the detailed proofs of the different embedding relations existing, on the one hand, between all the sublanguages of Dense Bach and, on the other hand, between the respective sublanguages of BachT and Dense Bach. We shall proceed to the end according to the logical introduction of primitives already used for studying the expressiveness of BachT and MRT. As a result, we first consider placing (by *tell*) some tokens on the store. Then we allow the sublanguages to question the state of the store by introducing *ask* and *nask* primitives. We thereafter allow to retrieve (by means of *get* primitives) some tokens. Finally, we study the combination of the removal of tokens with the

check for their presence and/or absence.

6.1.2 Adding tokens on the store

Proposition 37. $\mathcal{L}_{DB}$(tell) and $\mathcal{L}_{B}$(tell) are equivalent.

Proof. Indeed, thanks to proposition 36, $\mathcal{L}_{B}$(tell) $\leqslant \mathcal{L}_{DB}$(tell). Furthermore, by coding any $tell(t(m))$ primitive as m successive $tell(t)$ primitives, and by using the identity as decoder, one establishes that $\mathcal{L}_{DB}$(tell) $\leqslant \mathcal{L}_{B}$(tell). □

6.1.3 Checking for presence and/or absence when adding tokens

As a result of the expressiveness hierarchy [BJ98] (see Figure 3.5 in section 3.2.2), it also comes that both languages $\mathcal{L}_{B}$(ask,tell) and $\mathcal{L}_{B}$(nask,tell) are strictly more expressive than $\mathcal{L}_{DB}$(tell) since both have been established strictly more expressive than $\mathcal{L}_{B}$(tell). Let us now compare $\mathcal{L}_{B}$(ask,tell) with its dense counterpart.

Proposition 38. $\mathcal{L}_{B}$(ask,tell) $< \mathcal{L}_{DB}$(ask,tell)

Proof. **(i)** On the one hand, $\mathcal{L}_{B}$(ask,tell) $\leqslant \mathcal{L}_{DB}$(ask,tell) holds by proposition 36. **(ii)** On the other hand, $\mathcal{L}_{DB}$(ask,tell) $\not\leqslant \mathcal{L}_{B}$(ask,tell) may be established by contradiction. Consider $A = tell(t(1)); ask(t(2))$. As $\mathcal{O}(A) = \{(\{t(1)\}, \delta^{-})\}$, any computation of $\mathcal{C}(A)$ fails, by P_2 and P_3. However, it is possible to construct a successful computation for $\mathcal{C}(A)$. Indeed, $\mathcal{O}(tell(t(1)) \,;\, tell(t(1)) \,;\, ask(t(2))) = \{(\{t(2)\}, \delta^{+})\}$. Therefore any computation of $B = \mathcal{C}(tell(t(1)); tell(t(1)); ask(t(2)))$ starting on the empty store is successful, and hence so does any computation of $\mathcal{C}(tell(t(1)))$. Consider such a computation and let σ denote the final store. Given that $\mathcal{C}(tell(t(1))$ is composed of ask and tell primitives, it is possible to repeat the computation in order to deliver a successful computation for $\mathcal{C}(tell(t(1)) \,;\, tell(t(1)))$ ending in $\sigma \cup \sigma$ as final store. In view of the agent B and its successful computations, this computation can be continued by a successful computation for $\mathcal{C}(ask(t(2)))$. However, as $\mathcal{C}(ask(t(2)))$ is composed of ask and tell primitives only, this continuation succeeds also starting on σ (instead of $\sigma \cup \sigma$), which induces a successful computation for $\mathcal{C}(tell(t(1)); ask(t(2)))$. □

It is worth noting that the key point in the previous proof consists in the fact that the ingredients of the coding of *ask(t(2))*, namely *tell* and *ask* primitives, manipulate only one token at a time. Then putting only one token on the store in place of two has no influence on the action of *ask* and *tell* primitives, as they can select twice the same token. Hence this

absence of discrimination in the re-used token permits to generate a successful computation of the coded version of agent $tell(t(1))$; $ask(t(2))$, even if this agent must obviously fail.

It turns out that the same mechanism can be used to establish that $\mathcal{L}_{DB}$(ask,nask,tell) $\nleqslant$ $\mathcal{L}_{B}$(ask,nask,tell). To stress this similarity, this result is established now.

Proposition 39. $\mathcal{L}_{DB}$(ask,nask,tell) $\nleqslant$ $\mathcal{L}_{B}$(ask,nask,tell)

Proof. Similar to the second part of the proof of proposition 38 but, by using, $(tell(t(1)) \;||\; tell(t(1)))$; $ask(t(2))$, instead of $(tell(t(1)); tell(t(1)); ask(t(2)))$ to cope with the potential presence of nask primtives. □

Symmetrically to proposition 38, $\mathcal{L}_{B}$(nask,tell) is strictly less expressive than $\mathcal{L}_{DB}$(nask,tell).

Proposition 40. $\mathcal{L}_{B}$(nask,tell) $<$ $\mathcal{L}_{DB}$(nask,tell).

Proof. **(i)** On the one hand, $\mathcal{L}_{B}$(nask,tell) $\leqslant$ $\mathcal{L}_{DB}$(nask,tell) holds by proposition 36. **(ii)** On the other hand, $\mathcal{L}_{DB}$(nask,tell) $\nleqslant$ $\mathcal{L}_{B}$(nask,tell) may be obtained by contradiction. To that end, consider $tell(t(1)); nask(t(2))$. Since $\mathcal{O}(tell(t(1)) \;;\; nask(t(2))) = \{(\{t(1)\}, \delta^{+})\}$, any computation of $A = (\mathcal{C}(tell(t(1)) \;;\; \mathcal{C}(nask(t(2)))$ starting on the empty store is successful, by P_2 and P_3. As a result, any computation of $\mathcal{C}(tell(t(1)))$ succeeds. Consider one of them, say ending in the store σ. In view of agent A above, it can be continued by a successful computation, say C, of $\mathcal{C}(nask(t(2)))$. By duplicating each of its steps, one creates a successful computation for $D = (\mathcal{C}(tell(t(1))) \;||\; \mathcal{C}(tell(t(1))))$ ending in $\sigma \cup \sigma$. As $C = \mathcal{C}(nask(t(2)))$ is composed of nask and tell primitives only, D can be continued successfully by computation C, which yields a successful computation for $(\mathcal{C}(tell(t(1))) \;||\; \mathcal{C}(tell(t(1)))) \;;\; \mathcal{C}(nask(t(2)))$. Then by P_3, the contradiction comes from the fact that $((tell(t(1)) \;||\; tell(t(1))) \;;\; nask(t(2)))$ fails. □

As proposition 39 can be proved by the reasoning employed in proposition 38, so does $\mathcal{L}_{DB}$(nask,tell) $\nleqslant$ $\mathcal{L}_{B}$(ask,nask,tell) result from the reasoning used for previous proposition.

Proposition 41. $\mathcal{L}_{DB}$(nask,tell) $\nleqslant$ $\mathcal{L}_{B}$(ask,nask,tell)

Proof. By using the same reasoning as for the second part of proof of proposition 40 and by noting that the presence of the ask primitive in $\mathcal{L}_{B}$ does not destroy elements and so does not modify the state of the store σ. □

$\mathcal{L}_{DB}$(nask,tell) and $\mathcal{L}_{B}$(ask,tell) are not comparable with each other, as well as $\mathcal{L}_{DB}$(ask,tell) with regards to $\mathcal{L}_{B}$(nask,tell).

Proposition 42. $\mathcal{L}_{DB}(\text{nask,tell}) \wr \mathcal{L}_{B}(\text{ask,tell})$

Proof. **(i)** On the one hand, we have that $\mathcal{L}_{DB}(\text{nask,tell}) \not\leqslant \mathcal{L}_{B}(\text{ask,tell})$. Otherwise, by pattern 2 of transitivity and proposition 36, we have $\mathcal{L}_{B}(\text{nask,tell}) \leqslant \mathcal{L}_{B}(\text{ask,tell})$, which has been proved impossible in proposition 4.

(ii) On the other hand, one establishes that $\mathcal{L}_{B}(\text{ask,tell}) \not\leqslant \mathcal{L}_{DB}(\text{nask,tell})$ by contradiction. Consider $A = tell(t)\,;\,ask(t)$. One has $\mathcal{O}(A) = \{(\{t\}, \delta^{+})\}$. Hence, by P_3, $\mathcal{C}(A)$ succeeds whereas we shall establish that it has failing computations. Indeed, since $\mathcal{O}(ask(t)) = \{(\varnothing, \delta^{-})\}$, any computation of $\mathcal{C}(ask(t))$ starting on the empty store fails. As $\mathcal{C}(ask(t))$ is composed of nask and tell primitives, this can only occur by having a nask primitive preceded by a tell primitive. As enriching the initial content of the store does not change the behaviour of the *nask* primitive, any computation starting on any (arbitrary) store fails. As a consequence, even if $\mathcal{C}(tell(t))$ has a successful computation, this computation cannot be continued by a successful computation of $\mathcal{C}(ask(t))$. Consequently any computation of $\mathcal{C}(tell(t); ask(t))$ fails, which produces a contradiction. □

The next proposition establishes that $\mathcal{L}_{DB}(\text{ask,tell}) \wr \mathcal{L}_{B}(\text{nask,tell})$.

Proposition 43. $\mathcal{L}_{DB}(\text{ask,tell}) \wr \mathcal{L}_{B}(\text{nask,tell})$

Proof. **(i)** On the one hand, we have that $\mathcal{L}_{DB}(\text{ask,tell}) \not\leqslant \mathcal{L}_{B}(\text{nask,tell})$. Otherwise by pattern 2 of transitivity, $\mathcal{L}_{B}(\text{ask,tell}) \leqslant \mathcal{L}_{B}(\text{nask,tell})$ which has been proved impossible in proposition 4.

(ii) On the other hand, one establishes that $\mathcal{L}_{B}(\text{nask,tell}) \not\leqslant \mathcal{L}_{DB}(\text{ask,tell})$ by contradiction. Consider $A = tell(t)\,;\,nask(t)$. One has $\mathcal{O}(A) = \{(\{t\}, \delta^{-})\}$. By P_3, $\mathcal{C}(A)$ fails, whereas we shall establish that it has a successful computation. Indeed, since $\mathcal{O}(tell(t)) = \{(\{t\}, \delta^{+})\}$, any computation of $\mathcal{C}(tell(t))$ starting on the empty store is successful. Similarly, it follows from $\mathcal{O}(nask(t)) = \{(\varnothing, \delta^{+})\}$ that any computation of $\mathcal{C}(nask(t))$ starting on the empty store is successful, and, consequently, is any computation starting from any store, since $\mathcal{C}(nask(t))$ is composed of ask and tell primitives. Summing up, any (successful) computation of $\mathcal{C}(tell(t))$ starting on the empty store can be continued by a (successful) computation of $\mathcal{C}(nask(t))$, which leads to the contradiction. □

It is worth noting that the reasoning used in the second part of the proof will also be used to establish propositions 60 and 61.

$\mathcal{L}_{DB}$(nask,tell) and $\mathcal{L}_{DB}$(ask,tell) are not comparable with each other, as well as $\mathcal{L}_{DB}$(nask,tell) with regards to $\mathcal{L}_B$(ask,nask,tell).

Proposition 44. $\mathcal{L}_{DB}$(nask,tell) $\wr$ $\mathcal{L}_{DB}$(ask,tell)

Proof. On the one hand, we have that $\mathcal{L}_{DB}$(nask,tell) $\not\leqslant$ $\mathcal{L}_{DB}$(ask,tell). Otherwise by pattern 2 of transitivity, $\mathcal{L}_B$(nask,tell) $\leqslant$ $\mathcal{L}_{DB}$(ask,tell), which contradicts proposition 43. On the other hand, we have that $\mathcal{L}_{DB}$(ask,tell) $\not\leqslant$ $\mathcal{L}_{DB}$(nask,tell). Otherwise by pattern 2, $\mathcal{L}_B$(ask,tell) $\leqslant$ $\mathcal{L}_{DB}$(nask,tell), which contradicts proposition 42. □

Proposition 45. $\mathcal{L}_B$(ask,nask,tell) $\wr$ $\mathcal{L}_{DB}$(nask,tell)

Proof. On the one hand, we have that $\mathcal{L}_B$(ask,nask,tell) $\not\leqslant$ $\mathcal{L}_{DB}$(nask,tell). Otherwise by pattern 2 of transitivity $\mathcal{L}_B$(ask,tell) $\leqslant$ $\mathcal{L}_{DB}$(nask,tell), which contradicts proposition 42. On the other hand, proposition 41 establishes that $\mathcal{L}_{DB}$(nask,tell) $\not\leqslant$ $\mathcal{L}_B$(ask,nask,tell). □

$\mathcal{L}_B$(get,tell) and $\mathcal{L}_{DB}$(ask,tell) are not comparable with each other.

Proposition 46. $\mathcal{L}_B$(get,tell) $\wr$ $\mathcal{L}_{DB}$(ask,tell)

Proof. **(i)** On the one hand, one establishes that $\mathcal{L}_B$(get,tell) $\not\leqslant$ $\mathcal{L}_{DB}$(ask,tell) by contradiction. Consider $tell(t)$; $get(t)$. One has $\mathcal{O}(tell(t)\ ;\ get(t)) = \{(\varnothing, \delta^+)\}$. By P_2 and P_3, any computation of $\mathcal{O}(\mathcal{C}(tell(t))\ ;\ \mathcal{C}(get(t)))$ is thus successful. Since $\mathcal{C}(get(t))$ is composed of ask and tell primitives only and since ask and tell primitives do not destroy elements, at least one computation of $\mathcal{O}(\mathcal{C}(tell(t))\ ;\ \mathcal{C}(get(t))\ ;\ \mathcal{C}(get(t)))$ is successful. However, $\mathcal{O}(tell(t)\ ;\ get(t)\ ;\ get(t)) = \{(\varnothing, \delta^-)\}$, which provides the contradiction.

(ii) On the other hand, $\mathcal{L}_{DB}$(ask,tell) $\not\leqslant$ $\mathcal{L}_B$(get,tell) is established by contradiction. Consider $A = tell(t(1))\ ;\ (ask(t(2)) + tell(t(1)))$. One has $\mathcal{O}(A) = \{(\{t(2)\}, \delta^+)\}$. By P_3, any computation of $\mathcal{C}(A)$ thus succeeds. However it is possible to construct a failing computation. Indeed, let first observe that $\mathcal{O}(tell(t(1))\ ;\ tell(t(1))\ ;\ ask(t(2)))$ succeeds. Therefore by P_3 any computation of $B = \mathcal{C}(tell(t(1))\ ;\ tell(t(1))\ ;\ ask(t(2)))$ starting on the empty store is successful, and so does any computation of $\mathcal{C}(tell(t(1)))$. Consider such a computation C and let σ denote the final store. Given that $\mathcal{C}(tell(t(1)))$ is composed of get and tell primitives, it is possible to repeat the computation in order to deliver a successful computation for $\mathcal{C}(tell(t(1))\ ;\ tell(t(1)))$ ending in $\sigma \cup \sigma$ as final store. In view of agent B above, this computation can be continued by a successful computation C' for $\mathcal{C}(ask(t(2)))$. The first step s of C' is either a (single) tell which always succeeds or a (single) get which also succeeds on $\sigma \cup \sigma$, and therefore on σ. This leads to a first successful step s of $\mathcal{C}(ask(t(2)))$ after the computation C. As $(tell(t(1))\ ;\ ask(t(2)))$

fails, this computation prefix $C.s$ has only failing computations. Nevertheless, $C.s$ is a computation prefix of $\complement(tell(t(1)) \,;\, (ask(t(2)) \;+\; tell(t(1))))$, which leads to a failure and to the contradiction. □

The above proof can be repeated to deliver the four following propositions 47, 48, 49 and 50 as well as two propositions 62 and 63, of section 6.1.4.

Proposition 47. $\mathcal{L}_B$(nask,get,tell) $\nleqslant$ $\mathcal{L}_{DB}$(ask,nask,tell)

Proof. By using the reasoning of the first part of the proof of proposition 46 and by replacing the sequential composition of the two $get(t)$ primitives by a parallel one, in order to cope with the potential presence of nask primitives. □

Proposition 48. $\mathcal{L}_B$(get,tell) $\nleqslant$ $\mathcal{L}_{DB}$(ask,nask,tell)

Proof. By using the reasoning of the first part of the proof of proposition 46 and by replacing the sequential composition of the two $get(t)$ primitives by a parallel one, in order to cope with the potential presence of nask primitives. □

Proposition 49. $\mathcal{L}_{DB}$(ask,tell) $\nleqslant$ $\mathcal{L}_B$(nask,get,tell)

Proof. By employing the reasoning of the second part of the proof of proposition 46 where the sequential composition of $tell(t(1))$ with itself is replaced by a parallel one in order to cope with the potential presence of nask primitives. □

Proposition 50. $\mathcal{L}_{DB}$(ask,tell) $\nleqslant$ $\mathcal{L}_B$(ask,nask,tell)

Proof. By employing the reasoning of the second part of the proof of proposition 46 where the sequential composition of $tell(t(1))$ with itself is replaced by a parallel one in order to cope with the potential presence of nask primitives. □

To complete proposition 48, we now establish that $\mathcal{L}_{DB}$(ask,nask,tell) $\nleqslant$ $\mathcal{L}_B$(get,tell) leading to the result that $\mathcal{L}_B$(get,tell) $\wr$ $\mathcal{L}_{DB}$(ask,nask,tell).

Proposition 51. $\mathcal{L}_B$(get,tell) $\wr$ $\mathcal{L}_{DB}$(ask,nask,tell)

Proof. On the one hand, $\mathcal{L}_B$(get,tell) $\nleqslant$ $\mathcal{L}_{DB}$(ask,nask,tell) is established by proposition 48. On the other hand, $\mathcal{L}_{DB}$(ask,nask,tell) $\nleqslant$ $\mathcal{L}_B$(get,tell) is established by contradiction. Otherwise by the pattern 1 of sublanguage inclusion, $\mathcal{L}_{DB}$(ask,tell) $\leqslant$ $\mathcal{L}_{DB}$(ask,nask,tell) and then $\mathcal{L}_{DB}$(ask,tell) $\leqslant$ $\mathcal{L}_B$(get,tell) holds, which contradicts proposition 46. □

Similarly, to complete proposition 50, we now establish that $\mathcal{L}_B$(ask,nask,tell) $\nleqslant$ $\mathcal{L}_{DB}$(ask,tell) leading to the result that $\mathcal{L}_B$(ask,nask,tell) $\wr$ $\mathcal{L}_{DB}$(ask,tell).

Proposition 52. $\mathcal{L}_B$(ask,nask,tell) $\wr$ $\mathcal{L}_{DB}$(ask,tell)

Proof. On the one hand, $\mathcal{L}_{DB}$(ask,tell) $\nleqslant$ $\mathcal{L}_B$(ask,nask,tell) is established by proposition 50. On the other hand, $\mathcal{L}_B$(ask,nask,tell) $\nleqslant$ $\mathcal{L}_{DB}$(ask,tell) is established by contradiction. Otherwise by the pattern 1 of sublanguage inclusion, $\mathcal{L}_B$(nask,tell) $\leqslant$ $\mathcal{L}_B$(ask,nask,tell) and then $\mathcal{L}_B$(nask,tell) $\leqslant$ $\mathcal{L}_{DB}$(ask,tell) holds, which contradicts proposition 43. □

We now establish that $\mathcal{L}_{DB}$(ask,tell) is not comparable with $\mathcal{L}_B$(nask,get,tell).

Proposition 53. $\mathcal{L}_{DB}$(ask,tell) $\wr$ $\mathcal{L}_B$(nask,get,tell)

Proof. On the one hand, $\mathcal{L}_{DB}$(ask,tell) $\leqslant$ $\mathcal{L}_B$(nask,get,tell) is established by proposition 49. On the other hand, $\mathcal{L}_B$(nask,get,tell) $\nleqslant$ $\mathcal{L}_{DB}$(ask,tell) is established by contradiction. Otherwise by the pattern 2 of transitivity, $\mathcal{L}_B$(nask,tell) $\leqslant$ $\mathcal{L}_{DB}$(ask,tell) holds, which contradicts proposition 43. □

Let us now establish that $\mathcal{L}_{DB}$(nask,tell) and $\mathcal{L}_B$(ask,nask,tell) are strictly less expressive than $\mathcal{L}_{DB}$(ask,nask,tell).

Proposition 54. $\mathcal{L}_{DB}$(nask,tell) $<$ $\mathcal{L}_{DB}$(ask,nask,tell)

Proof. By sublanguage inclusion, one has $\mathcal{L}_{DB}$(nask,tell) $\leqslant$ $\mathcal{L}_{DB}$(ask,nask,tell). Moreover, if we had $\mathcal{L}_{DB}$(ask,nask,tell) $\leqslant$ $\mathcal{L}_{DB}$(nask,tell), then we would have $\mathcal{L}_{DB}$(ask,tell) $\leqslant$ $\mathcal{L}_{DB}$(nask,tell), which contradicts proposition 44. □

Proposition 55. $\mathcal{L}_B$(ask,nask,tell) $<$ $\mathcal{L}_{DB}$(ask,nask,tell)

Proof. It follows from proposition 36 that $\mathcal{L}_B$(ask,nask,tell) $\leqslant$ $\mathcal{L}_{DB}$(ask,nask,tell). Moreover, it follows from proposition 39 that $\mathcal{L}_{DB}$(ask,nask,tell) $\nleqslant$ $\mathcal{L}_B$(ask,nask,tell). □

$\mathcal{L}_{DB}$(ask,tell) is strictly less expressive than $\mathcal{L}_{DB}$(ask,nask,tell).

Proposition 56. $\mathcal{L}_{DB}$(ask,tell) $<$ $\mathcal{L}_{DB}$(ask,nask,tell)

Proof. On the one hand, $\mathcal{L}_{DB}$(ask,tell) $\leqslant$ $\mathcal{L}_{DB}$(ask,nask,tell) results from language inclusion. On the other hand, one has $\mathcal{L}_{DB}$(ask,nask,tell) $\nleqslant$ $\mathcal{L}_B$(ask,tell) since otherwise, by the pattern 2 of transitivity, $\mathcal{L}_{DB}$(nask,tell) $\leqslant$ $\mathcal{L}_{DB}$(ask,tell), which contradicts proposition 44. □

Symmetrically to proposition 53, $\mathcal{L}_B$(nask,get,tell) is not comparable with $\mathcal{L}_{DB}$(nask,tell).

Proposition 57. $\mathcal{L}_B$(nask,get,tell) $\wr$ $\mathcal{L}_{DB}$(nask,tell)

Proof. **(i)** On the one hand, $\mathcal{L}_B$(nask,get,tell) $\not\leqslant$ $\mathcal{L}_{DB}$(nask,tell). Otherwise, by the pattern 2 of transitivity, $\mathcal{L}_B$(ask,tell) $\leqslant$ $\mathcal{L}_{DB}$(nask,tell) which contradicts proposition 42.

(ii) On the other hand, $\mathcal{L}_{DB}$(nask,tell) $\not\leqslant$ $\mathcal{L}_B$(nask,get,tell) is established by contradiction. Consider $T2NoT = (tell(t(1)) \mid\mid tell(t(1))) \,;\, (nask(t(2)) \;+\; tell(t(1)))$. It has just a successful computation. Nevertheless we shall construct a failing computation for its coder. To that end, consider $T = \mathcal{C}(tell(t(1)))$. Given that $tell(t(1))$ succeeds, its coder T has only successful computations starting on the empty store. Consider one of them, say C, ending in the store σ. By repeating in turn each of its steps, it is possible to construct a successful computation, say CC, for $\mathcal{C}(tell(t(1)) \mid\mid tell(t(1)))$ ending in the store $\sigma \cup \sigma$. Consider now

$$\begin{aligned} T2N &= \mathcal{C}((tell(t(1)) \mid\mid tell(t(1))) \,;\, nask(t(2))) \\ &= (\mathcal{C}(tell(t(1))) \mid\mid \mathcal{C}(tell(t(1)))) \,;\, \mathcal{C}(nask(t(2))) \end{aligned}$$

As $tell(t(1)) \,;\, nask(t(2))$ succeeds, the computation C of $\mathcal{C}(tell(t(1)))$ can be continued by a successful computation for $\mathcal{C}(nask(t(2)))$. Consider such a computation and let s denote its first step. As C ends in the store σ, step s can also be successfully performed after CC, which ends in store $\sigma \cup \sigma$. However, $CC.s$ is a computation prefix for $T2N$, which, in view of the fact that $(tell(t(1)) \mid\mid tell(t(1))) \,;\, nask(t(2))$ fails, can only be continued by failing computations. However, these computations are also computations of $T2NoT$, which, thus provide the announced failing computation. □

$\mathcal{L}_B$(nask,get,tell) is not comparable with $\mathcal{L}_{DB}$(ask,nask,tell).

Proposition 58. $\mathcal{L}_{DB}$(ask,nask,tell) $\wr$ $\mathcal{L}_B$(nask,get,tell)

Proof. On the one hand, $\mathcal{L}_{DB}$(ask,nask,tell) $\not\leqslant$ $\mathcal{L}_B$(nask,get,tell) otherwise, by the pattern 3 of non embedding by transitivity, $\mathcal{L}_{DB}$(ask,tell) $\leqslant$ $\mathcal{L}_B$(nask,get,tell) which contradicts proposition 53. On the other hand, $\mathcal{L}_B$(nask,get,tell) $\not\leqslant$ $\mathcal{L}_{DB}$(ask,nask,tell) is established by proposition 47. □

$\mathcal{L}_{DB}$(nask,tell) is not comparable with $\mathcal{L}_B$(get,tell).

Proposition 59. $\mathcal{L}_{DB}$(nask,tell) $\wr$ $\mathcal{L}_B$(get,tell)

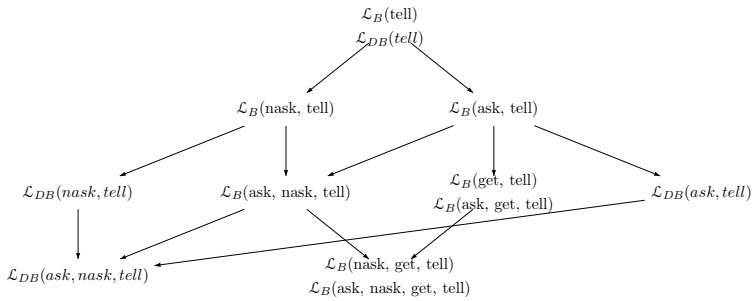


Figure 6.1: Embedding hierarchy of BachT and Dense Bach languages for the tell, ask and nask primitives in Dense Bach.

Proof. On the one hand, $\mathcal{L}_{DB}$(nask,tell) $\nleqslant$ $\mathcal{L}_B$(get,tell). Otherwise, by the pattern 2 of transitivity, $\mathcal{L}_B$(nask,tell) $\leqslant$ $\mathcal{L}_B$(get,tell) which contradicts proposition 8. On the other hand, $\mathcal{L}_B$(get,tell) $\nleqslant$ $\mathcal{L}_{DB}$(nask,tell). Otherwise $\mathcal{L}_B$(ask,tell) $\leqslant$ $\mathcal{L}_{DB}$(nask,tell) which contradicts proposition 42. □

In order to illustrate the results obtained so far, Figure 6.1 presents a synthesis of the expressive relations proved in the two previous subsections. Note that only the strict relations of expressiveness are shown: the absence of arrow between two languages means that there is no relation of expressiveness between them.

6.1.4 Retrieving tokens from the store

Let us now include the *get* primitive in the Dense Bach language. As previously announced the reasoning employed in the second part of the proof of proposition 43 can be used to establish the two following inequalities.

Proposition 60. $\mathcal{L}_B$(nask,tell) $\nleqslant$ $\mathcal{L}_{DB}$(get,tell)

Proof. Indeed, replacing ask by get in the Dense Bach language does not modify the successful behaviour of $\mathcal{C}(nask(t))$ not only when starting on the empty store, but also from any store. This still ensures that any successful computation of $\mathcal{C}(tell(t))$ can be followed by a successful computation of $\mathcal{C}(nask(t))$, leading to an obvious contradiction. □

Proposition 61. $\mathcal{L}_{DB}$(nask,get,tell) $\nleqslant$ $\mathcal{L}_{DB}$(get,tell)

Proof. The proof proceeds as the second part of the proof of proposition 43 by considering $tell(t(1))$ and $nask(t(1))$ instead of $tell(t)$ and $nask(t)$. Moreover, as noticed for proposition 60, the introduction of *get* primitives does not alter the reasoning. □

The two following propositions reuse the reasoning of proposition 46.

Proposition 62. $\mathcal{L}_{DB}$(get,tell) $\not\leqslant$ $\mathcal{L}_{DB}$(ask,tell)

Proof. By using the reasoning of the first part of the proof of proposition 46 to $get(t(1))$ and $tell(t(1))$ primitives. □

Proposition 63. $\mathcal{L}_{DB}$(get,tell) $\not\leqslant$ $\mathcal{L}_{B}$(get,tell)

Proof. By using the reasoning of the second part of the proof of proposition 46 in which the $ask(t(2))$ primitive is replaced by $get(t(2))$. □

We now prove that $\mathcal{L}_{DB}$(get,tell) and $\mathcal{L}_{DB}$(ask,get,tell) are equivalent.

Proposition 64. $\mathcal{L}_{DB}$(get,tell) = $\mathcal{L}_{DB}$(ask,get,tell)

Proof. On the one hand, one has $\mathcal{L}_{DB}$(get,tell) $\leqslant$ $\mathcal{L}_{DB}$(ask,get,tell) by pattern 1 of sublanguage inclusion. On the other hand, by coding $ask(t(m))$ as $get(t(m))$; $tell(t(m))$ and by using the identity as decoder, one has $\mathcal{L}_{DB}$(ask,get,tell) $\leqslant$ $\mathcal{L}_{DB}$(get,tell). □

Let us now establish that $\mathcal{L}_{B}$(get,tell) is strictly less expressive than $\mathcal{L}_{DB}$(get,tell).

Proposition 65. $\mathcal{L}_{B}$(get,tell) $<$ $\mathcal{L}_{DB}$(get,tell)

Proof. On the one hand, $\mathcal{L}_{B}$(get,tell) $\leqslant$ $\mathcal{L}_{DB}$(get,tell) holds by proposition 36. On the other hand, $\mathcal{L}_{DB}$(get,tell) $\not\leqslant$ $\mathcal{L}_{B}$(get,tell) is established in proposition 63. □

We now establish that $\mathcal{L}_{DB}$(ask,tell) is strictly less expressive than $\mathcal{L}_{DB}$(get,tell).

Proposition 66. $\mathcal{L}_{DB}$(ask,tell) $<$ $\mathcal{L}_{DB}$(get,tell).

Proof. On the one hand, $\mathcal{L}_{DB}$(ask,tell) $\leqslant$ $\mathcal{L}_{DB}$(get,tell) follows directly by coding $ask(t(m))$ as $get(t(m))$; $tell(t(m))$. On the other hand, $\mathcal{L}_{DB}$(get,tell) $\not\leqslant$ $\mathcal{L}_{DB}$(ask,tell) is established by proposition 62. □

We can now prove that $\mathcal{L}_{DB}$(get,tell) is not comparable with respect to $\mathcal{L}_{B}$(nask,tell), $\mathcal{L}_{DB}$(nask,tell), $\mathcal{L}_{B}$(nask,get,tell), $\mathcal{L}_{DB}$(ask,nask,tell) and $\mathcal{L}_{B}$(ask,nask,tell).

Proposition 67. $\mathcal{L}_{DB}(\text{get,tell}) \wr \mathcal{L}_{B}(\text{nask,tell})$

Proof. On the one hand, one has $\mathcal{L}_{DB}(\text{get,tell}) \nleqslant \mathcal{L}_{B}(\text{nask,tell})$. Indeed, otherwise, by the pattern 2 of transitivity $\mathcal{L}_{DB}(\text{ask,tell}) \leqslant \mathcal{L}_{DB}(\text{nask,tell})$ which contradicts proposition 44. On the other hand, $\mathcal{L}_{B}(\text{nask,tell}) \nleqslant \mathcal{L}_{DB}(\text{get,tell})$ is established by proposition 60. □

Proposition 68. $\mathcal{L}_{DB}(\text{get,tell}) \wr \mathcal{L}_{DB}(\text{nask,tell})$

Proof. On the one hand, $\mathcal{L}_{DB}(\text{get,tell}) \nleqslant \mathcal{L}_{DB}(\text{nask,tell})$ is established by contradiction. Indeed, otherwise, by pattern 2, $\mathcal{L}_{B}(\text{ask,tell}) \leqslant \mathcal{L}_{DB}(\text{nask,tell})$ which contradicts proposition 42. On the other hand, $\mathcal{L}_{DB}(\text{nask,tell}) \nleqslant \mathcal{L}_{DB}(\text{get,tell})$ is established similarly by contradiction. Otherwise, again by pattern 2, $\mathcal{L}_{B}(\text{nask,tell}) \leqslant \mathcal{L}_{DB}(\text{get,tell})$ which contradicts proposition 67. □

Proposition 69. $\mathcal{L}_{DB}(\text{get,tell}) \wr \mathcal{L}_{B}(\text{nask,get,tell})$

Proof. On the one hand, $\mathcal{L}_{DB}(\text{get,tell}) \nleqslant \mathcal{L}_{B}(\text{nask,get,tell})$ holds. Indeed, otherwise by the pattern 2 and with the result of proposition 66, we have $\mathcal{L}_{DB}(\text{ask,tell}) \leqslant \mathcal{L}_{B}(\text{nask,get,tell})$ which contradicts proposition 53. On the other hand, $\mathcal{L}_{B}(\text{nask,get,tell}) \nleqslant \mathcal{L}_{DB}(\text{get,tell})$ is established by contradiction. Indeed, otherwise by the pattern 3 $\mathcal{L}_{B}(\text{nask,tell}) \leqslant \mathcal{L}_{DB}(\text{get,tell})$ which contradicts proposition 67. □

Proposition 70. $\mathcal{L}_{DB}(\text{get,tell}) \wr \mathcal{L}_{DB}(\text{ask,nask,tell})$

Proof. On the one hand, $\mathcal{L}_{DB}(\text{get,tell}) \nleqslant \mathcal{L}_{DB}(\text{ask,nask,tell})$ is established by using the reasoning of the first part of the proof of proposition 46 to $get(t(1))$ and $tell(t(1))$ primitives, and by replacing the sequential composition of the two $get(t(1))$ primitives by a parallel one, in order to cope with the potential presence of nask primitives. On the other hand, $\mathcal{L}_{DB}(\text{ask,nask,tell}) \nleqslant \mathcal{L}_{DB}(\text{get,tell})$ holds. Otherwise by the pattern 2, $\mathcal{L}_{DB}(\text{nask,tell}) \leqslant \mathcal{L}_{DB}(\text{get,tell})$ which contradicts proposition 68. □

Proposition 71. $\mathcal{L}_{DB}(\text{get,tell}) \wr \mathcal{L}_{B}(\text{ask,nask,tell})$

Proof. On the one hand, $\mathcal{L}_{DB}(\text{get,tell}) \nleqslant \mathcal{L}_{B}(\text{ask,nask,tell})$ holds. Otherwise by pattern 2, $\mathcal{L}_{DB}(\text{get,tell}) \leqslant \mathcal{L}_{DB}(\text{ask,nask,tell})$ which contradicts proposition 70. On the other hand, $\mathcal{L}_{B}(\text{ask,nask,tell}) \nleqslant \mathcal{L}_{DB}(\text{get,tell})$ is directly established by contradiction. Otherwise, by the pattern 2, $\mathcal{L}_{B}(\text{nask,tell}) \leqslant \mathcal{L}_{DB}(\text{get,tell})$ which contradicts proposition 67. □

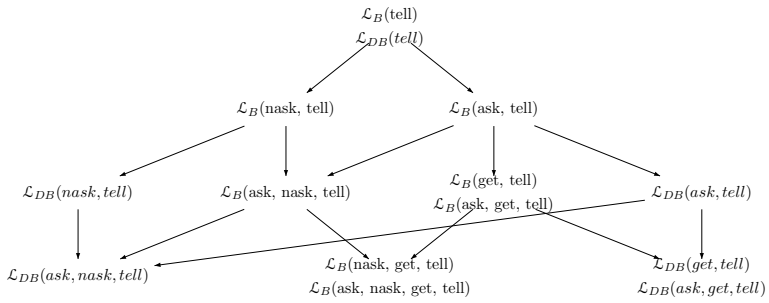


Figure 6.2: Embedding hierarchy of BachT and Dense Bach languages for the get primitive in Dense Bach.

Figure 6.2 adds the expressive relations related to the *get* primitive in Dense Bach to the figure obtained with the *tell*, *ask* and *nask* primitives.

6.1.5 Checking for the presence and/or absence when adding and/or retrieving tokens

$\mathcal{L}_{DB}$(nask,get,tell) and $\mathcal{L}_{DB}$(ask,nask,get,tell) are equivalent.

Proposition 72. $\mathcal{L}_{DB}$(nask,get,tell) = $\mathcal{L}_{DB}$(ask,nask,get,tell)

Proof. **(i)** On the one hand, $\mathcal{L}_{DB}$(nask,get,tell) $\leqslant$ $\mathcal{L}_{DB}$(ask,nask,get,tell) is established by language inclusion. **(ii)** On the other hand, to establish $\mathcal{L}_{DB}$(ask,nask,get,tell) $\leqslant$ $\mathcal{L}_{DB}$(nask,get,tell) we shall provide a coder such that the coding of the primitives $ask(t(n))$ and $nask(t(n))$ manipulate different tokens. To that end, as the set of tokens is enumerable, it is possible to associate each of them, say *t(n)*, with a pair $(t_1(n), t_2(n))$. Given such a coding of tokens, we define the compositional coder $\mathcal{C}$ as follows:

$$\begin{array}{lcl} \mathcal{C}(ask(t(n))) & = & get(t_2(n)) \,;\, tell(t_2(n)) \\ \mathcal{C}(nask(t(n))) & = & nask(t_1(n)) \\ \mathcal{C}(get(t(n))) & = & get(t_2(n)) \,;\, get(t_1(n)) \\ \mathcal{C}(tell(t(n))) & = & tell(t_1(n)) \,;\, tell(t_2(n)) \end{array}$$

The decoder $\mathcal{D}$ is defined as follows: $\mathcal{D}_{el}((\sigma, \delta)) = (\overline{\sigma}, \delta)$, where $\overline{\sigma}$ is composed of the tokens $t(n)$ for which $t_1(n)$ and $t_2(n)$ are in σ, the multiplicity of $t(n)$ being that of pairs $(t_1(n), t_2(n))$ in σ. □

$\mathcal{L}_{DB}$(ask,nask,tell) is strictly less expressive than $\mathcal{L}_{DB}$(ask,nask,get,tell), and is consequently less expressive than $\mathcal{L}_{DB}$(nask,get,tell), by proposition 72.

Proposition 73. $\mathcal{L}_{DB}(\text{ask,nask,tell}) < \mathcal{L}_{DB}(\text{ask,nask,get,tell})$

Proof. On the one hand, $\mathcal{L}_{DB}(\text{ask,nask,tell}) \leqslant \mathcal{L}_{DB}(\text{ask,nask,get,tell})$ results from language inclusion. On the other hand one has $\mathcal{L}_{DB}(\text{ask,nask,get,tell}) \nleqslant \mathcal{L}_{DB}(\text{ask,nask,tell})$. Otherwise by pattern 2 of transitivity, $\mathcal{L}_{DB}(\text{get,tell}) \leqslant \mathcal{L}_{DB}(\text{ask,nask,tell})$, which contradicts proposition 70. □

$\mathcal{L}_{DB}$(get,tell) is strictly less expressive than $\mathcal{L}_{DB}$(nask,get,tell).

Proposition 74. $\mathcal{L}_{DB}(\text{get,tell}) < \mathcal{L}_{DB}(\text{nask,get,tell})$

Proof. On the one hand, $\mathcal{L}_{DB}(\text{get,tell}) \leqslant \mathcal{L}_{DB}(\text{nask,get,tell})$ results from language inclusion. On the other hand, $\mathcal{L}_{DB}(\text{nask,get,tell}) \nleqslant \mathcal{L}_{DB}(\text{get,tell})$ results from proposition 61.

□

Finally, $\mathcal{L}_{B}$(ask,nask,get,tell) can be proved strictly less expressive than $\mathcal{L}_{DB}$(ask,nask,get,tell).

Proposition 75. $\mathcal{L}_{B}(\text{ask,nask,get,tell}) < \mathcal{L}_{DB}(\text{ask,nask,get,tell})$

Proof. **(i)** On the one hand, $\mathcal{L}_{B}(\text{ask,nask,get,tell}) \leqslant \mathcal{L}_{DB}(\text{ask,nask,get,tell})$ is directly deduced from proposition 36. **(ii)** On the other hand, using the pattern 2 of transitivity, if one had $\mathcal{L}_{DB}(\text{ask,nask,get,tell}) \leqslant \mathcal{L}_{B}(\text{ask,nask,get,tell})$ then $\mathcal{L}_{DB}(\text{get,tell}) \leqslant \mathcal{L}_{B}(\text{nask,get,tell})$ would hold, which contradicts proposition 69. □

Figure 6.3 presents the full expressive relations related to the *ask*, *nask*, *get* and *tell* primitives. It is worth observing that apart from $\mathcal{L}_{B}(\text{tell}) = \mathcal{L}_{DB}(\text{tell})$, any sublanguage of BachT is strictly less expressive than the corresponding sublanguage of Dense Bach. Moreover, the very nature of the *tell*, *ask*, *nask* and *get* primitives is kept by Dense Bach, which leads Dense Bach to share the sublanguage hierarchy of the sublanguages of BachT.

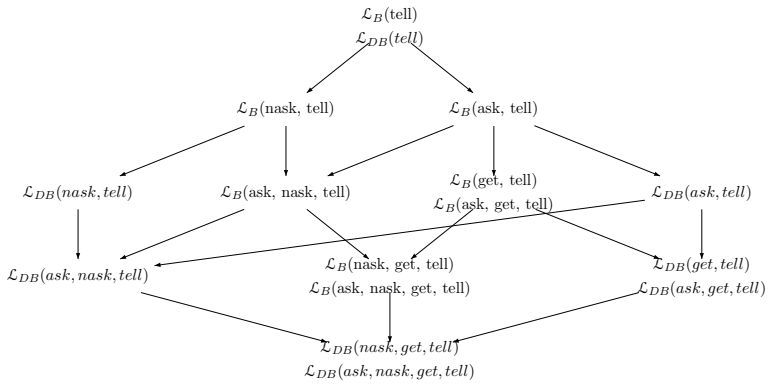


Figure 6.3: Embedding hierarchy of BachT and Dense Bach languages for all the primitives in Dense Bach.

6.2 Comparison with MRT

This section studies the expressiveness relations between the Dense Bach language $\mathcal{L}_{DB}$ and the multi-set rewriting language $\mathcal{L}_{MR}$. As for the study comparing Bach and Dense Bach, to treat relevant sublanguages, we shall restrict our attention to those that embody the *tell* primitive. In this comparison, only the relations of the expressiveness between the Dense Bach sublanguages and those for the multi-set rewriting language are taken into account, as the relations at the level of the language $\mathcal{L}_{MR}$ have already been established.

We shall also structure this section as the previous one. After generic results, we first start the exhaustive expressiveness comparison of the sublanguages that use the *tell* primitive to provide tokens in the tokenspace, and that question it about the presence or absence of tokens, respectively with the *ask* and *nask* primitives. Then we enrich the language with a *get* primitive, that permits to retrieve tokens from the tuplespace. Finally we consider all the languages that combine the get primitive with the ask and nask primitives. For this classification, the addition of the primitives is done with the point of view of the multi-set rewriting language. The text emphasizes the techniques of demonstration that are used in many propositions.

6.2.1 Generic patterns and results

Similarly to the comparison of BachT and Dense Bach, the three patterns permits to establish a number of propositions.

A first observation establishes that Dense Bach sublanguages are embedded in the corresponding multi-set rewriting sublanguages.

Proposition 76. $\mathcal{L}_{DB}(\ \chi) \leqslant \mathcal{L}_{MR}(\ \chi)$, for any subset of χ of primitives.

Proof. Immediate by defining the coder as follows:

$$\begin{array}{lcllcl} \mathcal{C}(tell(t(m))) & = & (\{\}, \{\underbrace{+t, +t, \cdots, +t}_{m\ times}\}) & \mathcal{C}(get(t(m))) & = & (\{\underbrace{+t, +t, \cdots, +t}_{m\ times}\}, \{\underbrace{-t, -t, \cdots, -t}_{m\ times}\}) \\ \mathcal{C}(ask(t(m))) & = & (\{\underbrace{+t, +t, \cdots, +t}_{m\ times}\}, \{\}) & \mathcal{C}(nask(t(m))) & = & (\{\underbrace{-t, -t, \cdots, -t}_{m\ times}\}, \{\}) \end{array}$$

and using the identity as decoder. □

6.2.2 Adding tokens on the store

When reduced to the tell primitive, Dense Bach and MRT are equivalent.

Proposition 77. $\mathcal{L}_{MR}(\text{tell}) = \mathcal{L}_{DB}(\text{tell})$

Proof. We have $\mathcal{L}_{DB}(\text{tell}) \leqslant \mathcal{L}_{MR}(\text{tell})$ by proposition 76. Furthermore, $\mathcal{L}_{MR}(\text{tell}) \leqslant \mathcal{L}_{DB}(\text{tell})$ is established by coding any tell primitive of $\mathcal{L}_{MR}(\text{tell})$ as the parallel composition of their dense versions: $\mathcal{C}(\{\}, \{\underbrace{+t_1, \cdots, +t_1}_{m_1\ times}, \cdots, \underbrace{+t_k, \cdots, +t_k}_{m_k\ times}\}) = \ ||_{i=1}^{k} tell(t_i(m_i))$. □

6.2.3 Checking for the presence and/or absence when adding tokens

Let us now consider the introduction of questioning the state of the store, regarding the presence or the absence of tokens on it. As a result of the expressiveness hierarchy obtained in section 6.1.3 and synthesized in Figure 6.3, it also comes that both languages $\mathcal{L}_{DB}$(ask,tell) and $\mathcal{L}_{DB}$(nask,tell) are strictly more expressive than $\mathcal{L}_{MR}$(tell) since both have been established strictly more expressive than $\mathcal{L}_{DB}$(tell). Let us now compare $\mathcal{L}_{DB}$(ask,tell) with its multiset counterpart.

Proposition 78. $\mathcal{L}_{DB}(\text{ask,tell}) < \mathcal{L}_{MR}(\text{ask,tell})$

Proof. **(i)** On the one hand, $\mathcal{L}_{DB}(\text{ask,tell}) \leqslant \mathcal{L}_{MR}(\text{ask,tell})$, by proposition 76. **(ii)** On the other hand, $\mathcal{L}_{MR}(\text{ask,tell}) \nleqslant \mathcal{L}_{DB}(\text{ask,tell})$ may be established by exploiting the inability of $\mathcal{L}_{DB}$(ask,tell) to atomically test the presence of two distinct tokens a and b. To do so, one considers $AB = (\{+a, +b\}, \{\})$ and assumes that $\mathcal{C}(AB)$ is in normal form and thus is written as $tell(\overline{t_1}); A_1 + \cdots + tell(\overline{t_p}); A_p + ask(\overline{u_1}); B_1 + \cdots + ask(\overline{u_q}); B_q$, where $\overline{t_i}$ and $\overline{u_j}$ denote the

token t_i and u_j associated with a density. In this normal form, we will establish that there is no alternative guarded by a $tell(\overline{t_i})$ operation and no alternative guarded by an $ask(\overline{u_j})$ operation either, which is impossible since $\mathcal{C}(AB)$ must contain at least one primitive.

Let us first establish by contradiction that there is no alternative guarded by a $tell(\overline{t_i})$ operation. Indeed, if there is an alternative guarded, say by $tell(\overline{t_i})$, then $D = \langle \mathcal{C}(AB)|\varnothing\rangle \rightarrow \langle A_i|\overline{t_i}\rangle$ is a valid computation prefix of $\mathcal{C}(AB)$. It should deadlock afterwards since $\mathcal{O}(AB) = \{(\varnothing, \delta^-)\}$. Let us denote by FP the failing computation. Now, as $\mathcal{C}(AB + (\{\}, \{+a\})) = \mathcal{C}(AB) + \mathcal{C}((\{\}, \{+a\}))$ by P_2, FP is also a valid computation prefix of $\mathcal{C}(AB + (\{\}, \{+a\}))$, and thus, as FP is a failing computation, $\mathcal{C}(AB + (\{\}, \{+a\}))$ admits a failing computation which, by P_3, contradicts the fact that $\mathcal{O}(AB + (\{\}, \{+a\})) = \{(\{a\}, \delta^+)\}$.

Secondly, we establish that there is also no alternative guarded by an $ask(\overline{u_j})$ operation. To that end, let us first consider two auxiliary computations. As $\mathcal{O}((\{\}, \{+a\})) = (\{a\}, \delta^+)$, any computation of $\mathcal{C}((\{\}, \{+a\})$ starting in the empty store succeeds. Let $\langle \mathcal{C}((\{\}, \{+a\}))|\varnothing\rangle \rightarrow \cdots \rightarrow \langle E|\{a_1, \ldots, a_m\}\rangle$ be such a computation. Similarly, let $\langle \mathcal{C}((\{\}, \{+b\}))|\varnothing\rangle \rightarrow \cdots \rightarrow \langle E|\{b_1, \ldots, b_n\}\rangle$ be one computation of $\mathcal{C}((\{\}, \{+b\}))$. The proof of the claim proceeds in two steps: none of the u_i's belong to $\{a_1, \ldots, a_m\} \cup \{b_1, \ldots, b_n\}$. First let us prove that none of the u_i's belong to $\{a_1, \ldots, a_m\}$. By contradiction, assume that $u_i = a_k$ for some k and that q is the density associated with u_i, namely, $\overline{u_i} = u_i(q)$. Let us observe that, since it is in $\mathcal{L}_{DB}$(ask,tell), the considered computation of $\mathcal{C}((\{\}, \{+a\}))$ can be repeated sequentially, as many times as needed. As a result, if, for an agent A and integer n, the notation A^n denotes the sequential composition of n copies of A and if for a token t, the notation t^n in a multiset denotes n occurrences of t, then $D' = \langle \mathcal{C}((\{\}, \{+a\})^q; AB)|\varnothing\rangle \rightarrow \cdots \rightarrow \langle \mathcal{C}(AB)|\{a_1^q, \ldots, a_m^q\}\rangle \rightarrow \langle B_j|\{a_1^q, \ldots, a_m^q\}\rangle$ is a valid computation prefix of $\mathcal{C}((\{\}, \{+a\})^q; AB)$, which can only be continued by failing suffixes. However D' induces the following computation prefix D'' for $(\{\}, \{+a\})^q; (AB + (\{+a\}, \{\}))$ which as just seen admits only successful computations: $D'' = \langle \mathcal{C}((\{\}, \{+a\})^q; (AB + (\{+a\}, \{\})))|\varnothing\rangle \rightarrow \cdots \rightarrow \langle \mathcal{C}(AB + (\{+a\}, \{\}))|\{a_1^q, \ldots, a_m^q\}\rangle \rightarrow \langle B_j|\{a_1^q, \ldots, a_m^q\}\rangle$. The proof proceeds similarly in the case $u_j \in \{b_1, \ldots, b_n\}$ for some $j \in \{1, \ldots, q\}$ by then considering $(\{\}, \{+b\})^p; AB$ and $(\{\}, \{+b\})^p; (AB + (\{+b\}, \{\}))$.

Finally, the fact that the $u_i's$ do not belong to $\{a_1, \ldots, a_m\} \cup \{b_1, \ldots, b_n\}$ induces a contradiction. Indeed, if this is the case then $\langle \mathcal{C}((\{\}, \{+a\}); (\{\}, \{+b\}); AB)|\varnothing\rangle \rightarrow \cdots \rightarrow \langle \mathcal{C}((\{\}, \{+b\}); AB)|\{a_1, \ldots, a_m\}\rangle \rightarrow \cdots \rightarrow \langle \mathcal{C}(AB)|\{a_1, \ldots, a_m, b_1, \ldots, b_n\}\rangle \nrightarrow$ is a valid failing computation prefix of $\mathcal{C}((\{\}, \{+a\}); (\{\}, \{+b\}); AB)$ whereas $(\{\}, \{+a\}); (\{\}, \{+b\}); AB$ has only one successful computation. □

Proof technique 3. The reasoning used to establish the above proposition is used once more to prove propositions 83, 85, 87, and 91, which respectively demonstrate that $\mathcal{L}_{MR}$(ask,tell) $\not\leqslant$ $\mathcal{L}_{DB}$(ask,nask,tell), $\mathcal{L}_{MR}$(ask,tell) $\not\leqslant$ $\mathcal{L}_{DB}$(get,tell), $\mathcal{L}_{MR}$(ask,tell) $\not\leqslant$ $\mathcal{L}_{DB}$(nask,get,tell) and $\mathcal{L}_{MR}$(get,tell) $\not\leqslant$ $\mathcal{L}_{DB}$(get,tell). Indeed, for each of these propositions, the intuition behind the

results is that the considered Dense Bach language is unable to atomically test the presence of two tokens or to remove those tokens. As a result, the same steps can be followed. An agent is first defined in a multi-set way to express the simultaneous testing of presence, or simultaneaous retrieval of some tokens. The coded version of this agent is then considered in a normal form and the proof technique consists in establishing by contradiction that actually none of the alternatives in this normal form exists. Hence the contradiction since by definition any agent is non-empty.

Symmetrically, $\mathcal{L}_{DB}$(nask,tell) is strictly less expressive than $\mathcal{L}_{MR}$(nask,tell). In order to establish this result, we use again lemma 1 of section 3.2.5.

Proposition 79. $\mathcal{L}_{DB}(\text{nask,tell}) < \mathcal{L}_{MR}(\text{nask,tell})$.

Proof. **(i)** On the one hand, $\mathcal{L}_{DB}(\text{nask,tell}) \leqslant \mathcal{L}_{MR}(\text{nask,tell})$ holds by proposition 76.

(ii) On the other hand, $\mathcal{L}_{MR}(\text{nask,tell}) \nleqslant \mathcal{L}_{DB}(\text{nask,tell})$ is proved by assuming the existence of a coder $\mathcal{C}$, and by establishing that it contains in fact no primitive, while it has to contain at least one. The proof proceeds as in proposition 78 but this time by exploiting the inability of $\mathcal{L}_{DB}$(nask,tell) to atomically test the absence of two distinct tokens a and b.

In the following, the construction of the tokens $\{a_1, \ldots, a_m\}$ and $\{b_1, \ldots, b_n\}$ associated with the coding of a and b will be generalized by the definition of a function $f : Stoken \rightarrow \mathcal{P}_f(Stoken)$, associating with each token a finite set of tokens. So for any token t, as $\mathcal{O}((\{\},\{+t\})) = \{(\{t\}, \delta^+)\}$, any computation of $\mathcal{C}((\{\},\{+t\}))$ starting in the empty store succeeds, let $\langle((\{\},\{+t\}))|\varnothing\rangle \rightarrow \cdots \rightarrow \langle E|\{t_1, \ldots, t_{m_t}\}\rangle$ be such a computation and let S_t denote the resulting store $\{t_1, \cdots t_{m_t}\}$.

Then the proof of the claim proceeds by examining two cases: (I) either there exist two tokens a and b such that $S_a \cap S_b = \varnothing$, (II) or $S_a \cap S_b \neq \varnothing$ for any pair of tokens a and b.

CASE I: Let us first suppose that there are two tokens a and b such that $S_a \cap S_b = \varnothing$. Consider $AB = (\{-a, -b\}, \{\})$ and $\mathcal{C}(AB)$ in its normal form:

$$tell(\overline{v_1}) \; ; A_1 + \cdots + tell(\overline{v_p}) \; ; A_p + nask(\overline{u_1}) \; ; B_1 + \cdots + nask(\overline{u_q}) \; ; B_q$$

The proof then proceeds by establishing that there are no alternatives guarded by $tell(\overline{v_i})$ nor by $nask(\overline{u_j})$. The absence of alternative guarded by a $tell(\overline{v_i})$ primitive is established as the proof of the second part of proposition 78: if this was not the case, then AB would point out a deadlocking computation for $(\{\},\{+a\}); (AB + (\{\},\{+a\}))$ which only admits successful computations. To prove the absence of an alternative guarded by a $nask(\overline{u_j})$ primitive, one establishes that the u_j's should belong to S_a and to S_b, which is impossible since $S_a \cap S_b = \varnothing$. By contradiction, assume that $u_j \notin S_a$ for some j (the case where $u_j \notin S_b$ is treated similarly). Then

$$\langle \mathcal{C}((\{\},\{+a\}) \; ; AB) \mid \varnothing\rangle \longrightarrow \cdots \longrightarrow \langle \mathcal{C}(AB) \mid S_a\rangle \longrightarrow \langle B_j \mid S_a\rangle$$

is a valid computation prefix of $\mathcal{C}((\{\},\{+a\})\,;AB)$ which can only be continued by failing suffixes. However, this prefix induces the following computation prefix D' for $\mathcal{C}((\{\},\{+a\})\,;(AB+(\{\},\{+a\}))$ which should only admit successful computations:

$$\langle \mathcal{C}((\{\},\{+a\})\,;(AB+(\{\},\{+a\}))) \mid \varnothing\rangle \longrightarrow \cdots$$

$$\longrightarrow \langle \mathcal{C}(AB+(\{\},\{+a\})) \mid S_a\rangle \longrightarrow \langle B_j \mid S_a\rangle$$

CASE II: Let us now suppose that $S_a \cap S_b \neq \varnothing$ for any pair of tokens a and b. As proved by Lemma 1, it is possible to construct an infinite sequence of distinct tokens x_i's and to identify an integer n such that

$$\bigcap_{i=1}^{n} S_{x_i} \neq \varnothing$$

and

$$\bigcap_{i=1}^{n} S_{x_i} = \bigcap_{i=1}^{n} S_{x_i} \cap S_{x_j}$$

for any $j > n$. Let us consider $NT = (\{-x_1,\cdots,-x_n\},\{\})$ and $\mathcal{C}(NT)$ in its normal form

$$tell(v_1)\,;A_1+\cdots+tell(v_p)\,;A_p+nask(u_1)\,;B_1+\cdots+nask(u_q)\,;B_q$$

By using a reasoning similar to the one employed for case I, one may prove that there are no alternatives guarded by a $tell(v_i)$ primitive and that $\{u_1,\cdots,u_q\} \subseteq S_{x_1}\cap\cdots\cap S_{x_n}$. Therefore $\mathcal{C}((\{\},\{+x_{n+1}\})\,;NT)$ has a failing computation since $S_{x_1}\cap\cdots\cap S_{x_n}\cap S_{x_{n+1}} = S_{x_1}\cap\cdots\cap S_{x_n}$ and thus $\{u_1,\cdots,u_q\} \subseteq S_{x_1}\cap\cdots\cap S_{x_n} \subseteq S_{x_{n+1}}$. However, this contradicts the fact that $(\{\},\{+x_{n+1}\})\,;NT$ has only one successful computation.

In conclusion, $\mathcal{C}(AB)$ reduces to an empty statement, which is not possible since it should contain at most one primitive. □

Proof technique 4. The reasoning used to establish the previous proposition is also used to prove propositions 82 and 88, where it is stated that $\mathcal{L}_{MR}$(nask,tell) $\nleq$ $\mathcal{L}_{DB}$(ask,nask,tell) and $\mathcal{L}_{MR}$(nask,tell) $\nleq$ $\mathcal{L}_{DB}$(nask,get,tell). Indeed, for these proofs, the reasoning amounts to consider the codings S_a and S_b of two tokens a and b and to distinguish whether they share or not a common element. In both cases, it is however proved that the coding of an agent has to be empty, which introduces a contradiction. Technically, it is worth noting that the presence of the *nask* primitive requires to consider a parallel composition instead of a sequential one in the coding of some primitives.

$\mathcal{L}_{MR}$(nask,tell) and $\mathcal{L}_{DB}$(ask,tell) are not comparable with each other, and so are $\mathcal{L}_{MR}$(ask,tell) and $\mathcal{L}_{DB}$(nask,tell).

Proposition 80. $\mathcal{L}_{MR}$(ask,tell) $\wr$ $\mathcal{L}_{DB}$(nask,tell)

Proof. On the one hand, we have that $\mathcal{L}_{MR}$(ask,tell) $\not\leqslant \mathcal{L}_{DB}$(nask,tell). Otherwise, by pattern 2 of transitivity, we have $\mathcal{L}_{DB}$(ask,tell) $\leqslant \mathcal{L}_{DB}$(nask,tell) which has been proved impossible in proposition 44. On the other hand, $\mathcal{L}_{DB}$(nask,tell) $\not\leqslant \mathcal{L}_{MR}$(ask,tell) is established by employing the same reasoning as the one used in the second part of the proof of proposition 43. □

Proposition 81. $\mathcal{L}_{MR}$(nask,tell) $\wr \mathcal{L}_{DB}$(ask,tell)

Proof. On the one hand, $\mathcal{L}_{MR}$(nask,tell) $\not\leqslant \mathcal{L}_{DB}$(ask,tell) holds. Otherwise, by pattern 2 of transitivity, we have $\mathcal{L}_{DB}$(nask,tell) $\leqslant \mathcal{L}_{DB}$(ask,tell) which has been proved impossible in proposition 44.

On the other hand, $\mathcal{L}_{DB}$(ask,tell) $\not\leqslant \mathcal{L}_{MR}$(nask,tell) is established by contradiction by considering $tell(t(1))\ ;\ ask(t(1)))$. Indeed, by noting that $\mathcal{O}(tell(t(1))\ ;\ ask(t(1))) = \{(\{t(1)\}, \delta^+)\}$, the reasoning developped in proposition 42 can be followed, which leads to the contradiction. □

We now prove that $\mathcal{L}_{DB}$(ask,nask,tell) and $\mathcal{L}_{MR}$(nask,tell) are not comparable with each other.

Proposition 82. $\mathcal{L}_{DB}$(ask,nask,tell) $\wr \mathcal{L}_{MR}$(nask,tell)

Proof. **(i)** We have that $\mathcal{L}_{DB}$(ask,nask,tell) $\not\leqslant \mathcal{L}_{MR}$(nask,tell). Otherwise, by pattern 2 of transitivity, $\mathcal{L}_{DB}$(ask,tell) $\leqslant \mathcal{L}_{MR}$(nask,tell), which has been proved impossible in proposition 81.

(ii) We have that $\mathcal{L}_{MR}$(nask,tell) $\not\leqslant \mathcal{L}_{DB}$(ask,nask,tell). The proof is an extension of the proof used in proposition 79 with normal forms extended with *ask* primitives.

Using the notations of this proof and following proof technique 4, we thus examine two cases and conclude for each one by a contradiction: (I) either there exist two tokens a and b such that $S_a \cap S_b = \varnothing$, (II) or $S_a \cap S_b \neq \varnothing$ for any pair of tokens a and b.

CASE I: Let us suppose that there are two tokens a and b such $S_a \cap S_b = \varnothing$. Consider $AB = (\{-a, -b\}, \{\})$ and $\mathcal{C}(AB)$ in its normal form:

$$\begin{array}{rl} & tell(\overline{t_1})\ ;\ A_1 + \cdots + tell(\overline{t_p})\ ;\ A_p \\ + & ask(\overline{u_1})\ ;\ B_1 + \cdots + ask(\overline{u_q})\ ;\ B_q \\ + & nask(\overline{v_1})\ ;\ C_1 + \cdots + nask(\overline{v_r})\ ;\ C_r \end{array}$$

The proof then proceeds by establishing that there are no alternatives guarded by *tell* and *nask* primitives. In that case, $\mathcal{C}(AB)$ reduces to

$$ask(\overline{u_1})\ ;\ B_1 + \cdots + ask(\overline{u_q})\ ;\ B_q$$

which thus fails on the empty store whereas $\mathcal{O}(AB) = \{(\varnothing, \delta^+)\}$, providing the contradiction. The absence of alternatives guarded by a $tell(\overline{t_i})$ primitive is established as the proof of the

second part of proposition 79: if this was not the case then AB would point out a deadlocking computation for $(\{\},\{+a\});(AB+(\{\},\{+a\}))$ which only admits successful computations. The absence of alternatives guarded by a $nask(\overline{v_i})$ primitive is established as the proof of the second part of proposition 79, namely by establishing that the v_i's should belong to S_a and to S_b, which is impossible since $S_a \cap S_b = \varnothing$.

CASE II: In the case where $S_a \cap S_b \neq \varnothing$ for any pair of tokens a and b, as proved by lemma 1, it is possible to construct an infinite sequence of distinct tokens x_i's and to identify an integer N such that

$$\bigcap_{i=1}^{N} S_{x_i} \neq \varnothing$$

and

$$\bigcap_{i=1}^{N} S_{x_i} = \bigcap_{i=1}^{N} S_{x_i} \cap S_{x_j}$$

for any $j > N$. Consider now $NT = (\{-x_1, \cdots, -x_n\}, \{\})$ and $\mathcal{C}(NT)$ in its normal form

$$\begin{array}{ll} & tell(\overline{t_1})\ ;\ A_1 + \cdots + tell(\overline{t_p})\ ;\ A_p \\ + & ask(\overline{u_1})\ ;\ B_1 + \cdots + ask(\overline{u_q})\ ;\ B_q \\ + & nask(\overline{v_1})\ ;\ C_1 + \cdots + nask(\overline{v_r})\ ;\ C_r \end{array}$$

By reasoning similarly to case I, one may prove that there are no alternatives guarded by a $tell(t_i)$ primitive. As regards the ask and nask primitives, the proof proceeds by contradiction and establishes successively that

$$\{u_1, \cdots, u_q\} \cap (S_{x_1} \cup \cdots \cup S_{x_n}) = \varnothing,$$

and that

$$\{v_1, \cdots, v_r\} \subseteq (S_{x_1} \cap \cdots \cap S_{x_n})$$

and derive a contradiction therefrom.

Let us first establish that $\{u_1, \cdots, u_q\} \cap (S_{x_1} \cup \cdots \cup S_{x_n}) = \varnothing$. By contradiction, assume $u_j \in S_{x_i}$, for some i, j and let $\overline{u_j} = u_j(p)$. By using p instances of $\mathcal{C}((\{\},\{+x_i\}))$ and repeating in turn in each of the instances the step of $\mathcal{C}((\{\},\{+x_i\}))$, one may produce p duplications of S_{x_i}. Consequently,

$$F = \langle (||_{k=1}^{p} \mathcal{C}((\{\},\{+x_i\}))) \ ;\ \mathcal{C}(NT) \mid \varnothing \rangle \longrightarrow \cdots$$

$$\longrightarrow \langle \mathcal{C}(NT) \mid \cup_{k=1}^{p} S_{x_i} \rangle \longrightarrow \langle B_j \mid \cup_{k=1}^{p} S_{x_i} \rangle$$

is a valid computation prefix for $(||_{k=1}^{p} \mathcal{C}((\{\},\{+x_i\}))) \ ;\ \mathcal{C}(NT))$ which can only be continued by failing suffixes. However, F is also a computation prefix for $\mathcal{C}((\{\},\{+x_i\}) \ ;\ (NT + (\{+x_i\},\{\})))$ which thus induces a failing computation for it whereas $(\{\},\{+x_i\}) \ ;\ (NT + (\{+x_i\},\{\})))$ has only one successful computation.

Let us now establish that $\{v_1, \cdots, v_r\} \subseteq (S_{x_1} \cap \cdots \cap S_{x_n})$. Assume $v_k \notin S_{x_i}$, for some k, i. Then

$$F' = \langle \mathcal{C}((\{\}, \{+x_i\})) \; ; \; \mathcal{C}(NT) \mid \varnothing \rangle \longrightarrow \cdots \longrightarrow \langle \mathcal{C}(NT) \mid S_{x_i} \rangle \longrightarrow \langle C_k \mid S_{x_i} \rangle$$

is a valid computation prefix for $\mathcal{C}((\{\}, \{+x_i\}) \; ; \; NT)$ which, can only be continued by failing suffixes. However, it is also a computation prefix for $(\{\}, \{+x_i\}) \; ; \; (NT + (\{+x_i\}, \{\})))$, for which it thus induces a failing computation whereas $(\{\}, \{+x_i\}) \; ; \; (NT \; + \; (\{+x_i\}, \{\})))$ has only one successful computation.

In order to establish the final contradiction, let us consider $\mathcal{C}((\{\}, \{+x_{n+1}\}) \; ; \; NT)$. A possible computation prefix is as follows:

$$\langle \mathcal{C}((\{\}, \{+x_{n+1}\})) \; ; \; \mathcal{C}(NT) \mid \varnothing \rangle \longrightarrow \cdots \longrightarrow \langle \mathcal{C}(NT) \mid S_{x_{n+1}} \rangle.$$

Since $(\{\}, \{+x_{n+1}\}) \; ; \; NT$ has a successful computation and since, $\{v_1, \cdots, v_r\} \subseteq S_{x_1} \cap \cdots \cap S_{x_n} \subseteq S_{x_{n+1}}$ by the choice of the x_i's and the above inclusion, one may think of excluding the execution of a *nask* primitive and therefore of forcing the execution of an *ask* primitive and thus the existence of a j such that $u_j \in S_{x_{n+1}}$. This is almost true except for the density of tokens, which may lead a *nask* primitive to succeed even if the corresponding token is present on the blackboard. This problem can be circumvented by using d parallel compositions of $(\{\}, \{+x_{n+1}\})$, where d is the maximal density of the tokens in the primitives $nask(\overline{v_1})$, $\ldots$,$nask(\overline{v_r})$. This thus leads to consider $\mathcal{C}((||_{k=1}^{d}(\{\}, \{+x_{n+1}\})) \; ; \; NT)$, for which the following is a valid computation prefix

$$\langle \mathcal{C}((||_{k=1}^{d}(\{\}, \{+x_{n+1}\}))) \; ; \; \mathcal{C}(NT) \mid \varnothing \rangle \longrightarrow \cdots \longrightarrow \langle \mathcal{C}(NT) \mid \cup_{k=1}^{d} S_{x_{n+1}} \rangle.$$

and which induces the existence of some j such that $u_j \in S_{x_{n+1}}$. Now consider $(||_{k=1}^{d}(\{\}, \{+x_{n+1}\})) \; ; \; (\{\}, \{+x_1\}) \; ; \; NT$ which has failing computations only. At the coded level, since $\mathcal{L}_{DB}$(ask,nask,tell) does not contain any destructive primitive, the computation of $\mathcal{C}((\{\}, \{+x_1\}))$ can only enrich the store resulting from the d parallel computations of $\mathcal{C}((\{\}, \{+x_{n+1}\}))$, say by some set of tokens σ. Consequently, the following derivation sequence G is a valid computation prefix for $\mathcal{C}((||_{k=1}^{d}(\{\}, \{+x_{n+1}\})) \; ; \; (\{\}, \{+x_1\}) \; ; \; NT)$, which should be continued by failing suffixes only:

$$\begin{aligned} G \;\; = \;\; & \langle \mathcal{C}((||_{k=1}^{d}(\{\}, \{+x_{n+1}\}))) \; ; \; \mathcal{C}((\{\}, \{+x_1\})) \; ; \; \mathcal{C}(NT) \mid \varnothing \rangle \longrightarrow \cdots \\ & \longrightarrow \langle \mathcal{C}((\{\}, \{+x_1\})) \; ; \; \mathcal{C}(NT) \mid \cup_{k=1}^{d} S_{x_{n+1}} \rangle \longrightarrow \cdots \\ & \longrightarrow \langle \mathcal{C}(NT) \mid (\cup_{k=1}^{d} S_{x_{n+1}}) \cup \sigma \rangle \longrightarrow \langle B_j \mid (\cup_{k=1}^{d} S_{x_{n+1}}) \cup \sigma \rangle \end{aligned}$$

However, G is also a computation prefix G' for $\mathcal{C}((||_{k=1}^{d}(\{\}, \{+x_{n+1}\})) \; ; \; (\{\}, \{+x_1\}) \; ; \; (NT \; + \; (\{+x_{n+1}\}, \{\})))$, which thus induces a failing computation for $(||_{k=1}^{d}(\{\}, \{+x_{n+1}\})) \; ; \; (\{\}, \{+x_1\}) \; ; \; (NT \; + \; (\{+x_{n+1}\}, \{\})))$ which is impossible since it admits only a successful computation. □

Let us now prove that $\mathcal{L}_{MR}$(ask,tell) is not comparable with $\mathcal{L}_{DB}$(ask,nask,tell).

Proposition 83. $\mathcal{L}_{MR}$(ask,tell) $\wr$ $\mathcal{L}_{DB}$(ask,nask,tell)

Proof. **(i)** Otherwise, $\mathcal{L}_{DB}$(nask,tell) $\leqslant$ $\mathcal{L}_{DB}$(ask,nask,tell) $\leqslant$ $\mathcal{L}_{MR}$(ask,tell) which has been proved impossible in proposition 80.

(ii) Let us proceed by contradiction by assuming the existence of a coder $\mathcal{C}$. Let us fix two distinct tokens a and b. Let n be the cumulative occurrences of tokens in the *nask* primitives of $\mathcal{C}((\{\},\{+a\}))$.

As $\mathcal{C}((\{\},\{+a\}))$ has only successful computations, let, as in the proof of the second part of proposition 79, S_a be the store resulting from one of them. As $(||_{k=1}^{n+2}(\{\},\{+b\}))$; $(\{\},\{+a\}))$ succeeds as well, let S'_b denote the store resulting from one successful computation of its coding. Consider finally $ABs = (\{+a,+b,\cdots,+b\},\{\})$ requesting one a with $n+3$ copies of b and $\mathcal{C}(ABs)$ in its normal form:

$$\begin{array}{rl} & tell(\overline{t_1})\ ;\ A_1 + \cdots + tell(\overline{t_p})\ ;\ A_p \\ + & ask(\overline{u_1})\ ;\ B_1 + \cdots + ask(\overline{u_q})\ ;\ B_q \\ + & nask(\overline{v_1})\ ;\ C_1 + \cdots + nask(\overline{v_r})\ ;\ C_r \end{array}$$

We shall establish, as explained in proof technique 3, (I) that there are no alternatives guarded by $tell(\overline{t_i})$ and $nask(\overline{v_j})$ primitives, and (II) that $\{u_1,\cdots,u_q\} \cap (S_a \cup S'_b) = \varnothing$. Assuming these facts proved, as repeating once more $\mathcal{C}((\{\},\{+b\}))$ just add copies of tokens already present in $S_a \cup S'_b$, it follows that $\mathcal{C}(||_{k=1}^{n+3}(\{\},\{+b\}))$; $(\{\},\{+a\})$; $ABs)$ fails, which is absurd by P_3, since, by construction, $||_{k=1}^{n+3}(\{\},\{+b\}))$; $(\{\},\{+a\})$; ABs has one successful computation. Hence the announced contradiction.

STEP I: As for proposition 79, the proof establishes, by contradiction, that there are neither alternatives guarded by a *tell* primitive, nor alternatives guarded by a *nask* primitive. Indeed assuming respectively the existence of a $tell(\overline{t_i})$; A_i alternative or a $nask(\overline{v_i})$; C_i alternative points out in both cases a failing computation for $\mathcal{C}(AB + (\{\},\{+a\}))$. This clearly contradicts the fact that $\mathcal{O}(AB + (\{\},\{+a\})) = (\{a\},\delta^+)$.

STEP II: Let us now prove that $\{u_1,\cdots,u_q\} \cap (S_a \cup S'_b) = \varnothing$. This is established in two steps by demonstrating (1) that $\{u_1,\cdots,u_q\} \cap S_a = \varnothing$, and (2) that $\{u_1,\cdots,u_q\} \cap S'_b = \varnothing$.

First let us prove that $\{u_1,\cdots,u_q\} \cap S_a = \varnothing$. Assume $u_i \in S_a$ and let q be the density associated with u_i, namely, $\overline{u_i} = u_i(q)$. Let us observe that each step of the considered computation of $\mathcal{C}((\{\},\{+a\}))$ can be repeated in turn, in as many parallel occurences of it as needed, so that

$$\begin{array}{rl} P = \langle \mathcal{C}((||_{k=1}^{q}(\{\},\{+a\}))\ ;\ ABs) | \varnothing \rangle & \\ \rightarrow \cdots \rightarrow & \langle \mathcal{C}(ABs) | \cup_{k=1}^{q} S_a \rangle \\ \rightarrow & \langle B_i | (\cup_{k=1}^{q} S_a) \rangle \end{array}$$

is a valid computation prefix of $\mathcal{C}((||_{k=1}^{q}(\{\},\{+a\}))\,;ABs)$, which can only be continued by failing suffixes. However P induces the following computation prefix P' for $\mathcal{C}((||_{k=1}^{q}(\{\},\{+a\}))\,;(ABs+(\{\},\{+a\})))$ which admits only successful computations:

$$\begin{aligned} P' = \langle \mathcal{C}((||_{k=1}^{q}(\{\},\{+a\}))\,;(ABs+(\{\},\{+a\})))|\varnothing\rangle \\ \rightarrow\cdots\rightarrow \quad \langle \mathcal{C}(ABs+(\{\},\{+a\}))|\cup_{k=1}^{q} S_a\rangle \\ \rightarrow \quad \langle B_i|(\cup_{k=1}^{q} S_a)\rangle \end{aligned}$$

Hence the contradiction.

Secondly, the proof that $\{u_1,\cdots,u_q\}\cap S'_b=\varnothing$ is established similarly by considering S'_b instead of S_a and $(\{\},\{+b\})$ instead of $(\{\},\{+a\})$. □

We now prove that $\mathcal{L}_{DB}$(ask,nask,tell) is stricly less expressive than $\mathcal{L}_{MR}$(ask,nask,tell).

Proposition 84. $\mathcal{L}_{DB}$(ask,nask,tell) $<$ $\mathcal{L}_{MR}$(ask,nask,tell)

Proof. **(i)** On the one hand, the fact that $\mathcal{L}_{DB}$(ask,nask,tell) $\leqslant$ $\mathcal{L}_{MR}$(ask,nask,tell) is immediate by proposition 76. **(ii)** On the other hand, $\mathcal{L}_{MR}$(ask,nask,tell) $\nleqslant$ $\mathcal{L}_{DB}$(ask,nask,tell). Otherwise, by pattern 2 of transitivity, from $\mathcal{L}_{MR}$(nask,tell) $\leqslant$ $\mathcal{L}_{MR}$(ask,nask,tell), one would get $\mathcal{L}_{MR}$(nask,tell) $\leqslant$ $\mathcal{L}_{DB}$(ask,nask,tell), which has been proved impossible in proposition 82. □

Proposition 85. $\mathcal{L}_{DB}$(get,tell) $\wr$ $\mathcal{L}_{MR}$(ask,tell)

Proof. **(i)** The reasoning used to prove that $\mathcal{L}_{DB}$(get,tell) $\nleqslant$ $\mathcal{L}_{MR}$(ask,tell) is the same as the one used in the first part of the proof of proposition 46. It works by contradiction and by establishing that $tell(t(1))\,;get(t(1))$ can produce a successful computation for $\mathcal{C}(tell(t(1)))\,;\mathcal{C}(get(t(1)))\,;\mathcal{C}(get(t(1)))$, which can obviously not be the case.

(ii) Intuitively, $\mathcal{L}_{DB}$(get,tell) is unable to atomically test the presence of a and b. Let us thus consider $AB=(\{+a,+b\},\{\})$ and assume that $\mathcal{C}(AB)$ is in normal form and thus can be rewritten as

$$tell(\overline{t_1});A_1+\cdots+tell(\overline{t_p});A_p+get(\overline{u_1});B_1+\cdots+get(\overline{u_q});B_q$$

where $\overline{t_i}$ and $\overline{u_j}$ denote the token t_i and u_j associated with a density.

The proof proceeds as explained in proof technique 3 by establishing (I) that there is no alternative guarded by a $tell(\overline{t_i})$ operation, and (II) that there is no alternative guarded by a $get(\overline{u_j})$ operation, in which case, $\mathcal{C}(AB)$ is equivalent to an empty statement, which is impossible since it is composed of at least one primitive.

STEP I: Let us first establish that there is no existence of an alternative guarded by a $tell(\overline{t_i})$ operation. Otherwise it would point out a failing computation for $\mathcal{C}(AB + (\{\}, \{+a\}))$, contradicting the fact that $\mathcal{O}(AB + (\{\}, \{+a\})) = (\{a\}, \delta^+)$.

STEP II: Let us now establish that there is no alternative guarded by a $get(\overline{u_j})$ operation. To that end, let us first consider two auxiliary computations. As $\mathcal{O}((\{\}, \{+a\})) = \{(\{a\}, \delta^+)\}$, any computation of $\mathcal{C}((\{\}, \{+a\}))$ starting in the empty store succeeds. Let $\langle((\{\}, \{+a\}))|\varnothing\rangle \rightarrow \cdots \rightarrow \langle E|\{a_1, \ldots, a_m\}\rangle$ be such a computation. Similarly, let $\langle((\{\}, \{+b\}))|\varnothing\rangle \rightarrow \cdots \rightarrow \langle E|\{b_1, \ldots, b_n\}\rangle$ be one computation of $\mathcal{C}((\{\}, \{+b\}))$.

The proof of the claim proceeds in two steps as for proposition 78: none of the u_i's belong to $\{a_1, \ldots, a_m\} \cup \{b_1, \ldots, b_n\}$ but, in that case, a contradiction occurs from the analysis of $\mathcal{C}((\{\}, \{+a\}); (\{\}, \{+b\}); AB)$. As a result, none of the u_i's exist, namely there is no alternative guarded by a $get(\overline{u_j})$ operation. □

We have that $\mathcal{L}_{DB}$(get,tell) is not comparable with $\mathcal{L}_{MR}$(ask,nask,tell).

Proposition 86. $\mathcal{L}_{DB}$(get,tell) $\wr$ $\mathcal{L}_{MR}$(ask,nask,tell)

Proof. On the one hand, $\mathcal{L}_{DB}$(get,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(ask,nask,tell). The proof proceeds as for establishing that $\mathcal{L}_{DB}$(get,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(ask,tell) (see proposition 85) by considering $tell(t(1))\,;\, get(t(1))$ and $tell(t(1))\,;\, (get(t(1)) \;||\; get(t(1)))$, the parallel composition $\mathcal{C}(get(t(1))) \;||\; \mathcal{C}(get(t(1)))$ repeating in turn each step of $\mathcal{C}(get(t(1)))$.

On the other hand, $\mathcal{L}_{MR}$(ask,nask,tell) $\not\leqslant$ $\mathcal{L}_{DB}$(get,tell). Otherwise, by pattern 2, one would have that $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_{MR}$(ask,nask,tell) $\leqslant$ $\mathcal{L}_{DB}$(get,tell) which has been proved impossible in proposition 85. □

We now prove that $\mathcal{L}_{MR}$(ask,tell) is not comparable with $\mathcal{L}_{DB}$(nask,get,tell).

Proposition 87. $\mathcal{L}_{DB}$(nask,get,tell) $\wr$ $\mathcal{L}_{MR}$(ask,tell)

Proof. **(i)** On the one hand, $\mathcal{L}_{DB}$(nask,get,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(ask,tell). Otherwise, by the pattern 3 of non embedding by transitivity, $\mathcal{L}_{DB}$(nask,tell) $\leqslant$ $\mathcal{L}_{MR}$(ask,tell) which has been proved impossible in the proof of the second part of proposition 80.

(ii) The intuition behind the proof is again that $\mathcal{L}_{DB}$(nask,get,tell) is not able to test atomically the presence of two distinct tokens a and b. We then proceed by contradiction using these two tokens. However, the destructive character of *get* primitives coupled to the test for absence of *nask* slightly complicate our task of producing a contradiction. To that end, we shall "saturate" their effect by taking as many instances of codings in parallel and thereby by extending the sets S_b introduced in the proof of the second part of proposition 79.

Let us thus proceed by contradiction by assuming the existence of a coder $\mathcal{C}$. Take two distinct tokens a and b. Let n be the cumulative sum of the densities associated with the *nask* and *get* primitives of $\mathcal{C}((\{\},\{+a\}))$. As $\mathcal{C}((\{\},\{+a\}))$ has only successful computations, let, as in the proof of the second part of proposition 79, S_a be the store resulting from one of them. As $(||_{k=1}^{n+2}(\{\},\{+b\}))\ ;\ (\{\},\{+a\}))$ succeeds as well, let S'_b denote the store resulting from one successful computation of its coding. Consider finally $ABs = (\{+a,+b,\cdots,+b\},\{\})$ requesting one a with $n+3$ copies of b and $\mathcal{C}(ABs)$ in its normal form:

$$\begin{array}{l} tell(\overline{t_1})\ ;\ A_1 + \cdots + tell(\overline{t_p})\ ;\ A_p \\ + get(\overline{u_1})\ ;\ B_1 + \cdots + get(\overline{u_q})\ ;\ B_q \\ + nask(\overline{v_1})\ ;\ C_1 + \cdots + nask(\overline{v_r})\ ;\ C_r \end{array}$$

Following proof technique 3, we shall establish (I) that there are no alternatives guarded by *tell* and *nask* primitives, and (II) that $\{u_1,\cdots,u_q\} \cap (S_a \cup S'_b) = \varnothing$. Assuming these two points proved, a contradiction can be produced as follows. Indeed, in view of the saturation provided by the $n+2$ copies of $\mathcal{C}((\{\},\{+b\}))$, adding one more, only adds tokens present in $S_a \cup S'_b$. As a result, $\mathcal{C}(||_{k=1}^{n+3}(\{\},\{+b\}))\ ;\ (\{\},\{+a\})\ ;\ ABs)$ fails whereas $||_{k=1}^{n+3}(\{\},\{+b\}))\ ;\ (\{\},\{+a\})\ ;\ ABs$ has only one successful computation. Hence the contradiction.

STEP I: Let us first establish that there are no alternative guarded by a $tell(\overline{t_i})$ primitive. The proof proceeds by contradiction as in the proof of the second part of proposition 78, by pointing out a failing computation for $\mathcal{C}(AB + (\{\},\{+a\}))$, contradicting the fact that $\mathcal{O}(AB + (\{\},\{+a\})) = (\{a\},\delta^+)$.

In a similar way there are no alternative guarded by a *nask* primitive. Indeed assuming the existence of a $nask(\overline{v_i})\ ;\ C_i$ alternative again points out a failing computation for $\mathcal{C}(AB + (\{\},\{+a\}))$, contradicting the fact that $\mathcal{O}(AB + (\{\},\{+a\})) = (\{a\},\delta^+)$.

STEP II: Let us now establish that $\{u_1,\cdots,u_q\} \cap (S_a \cup S'_b) = \varnothing$. This is proved in two steps by establishing (1) that $\{u_1,\cdots,u_q\} \cap S_a = \varnothing$, and (2) that $\{u_1,\cdots,u_q\} \cap S'_b = \varnothing$.

First we have that $\{u_1,\cdots,u_q\} \cap S_a = \varnothing$. By contradiction, assume that $u_i \in S_a$ for some i and let q be the density associated with u_i, namely, $\overline{u_i} = u_i(q)$. Let us observe that each step of the considered computation of $\mathcal{C}((\{\},\{+a\}))$ can be repeated in turn, in as many parallel occurences of it as needed, so that

$$\begin{array}{rl} P = \langle \mathcal{C}((||_{k=1}^{q}(\{\},\{+a\}))\ ;\ ABs) | \varnothing \rangle & \\ \rightarrow \cdots \rightarrow & \langle \mathcal{C}(ABs) |\ \cup_{k=1}^{q} S_a \rangle \\ \rightarrow & \langle B_i | (\cup_{k=1}^{q} S_a) \backslash \{\overline{u_i}\} \rangle \end{array}$$

is a valid computation prefix of $\mathcal{C}((||_{k=1}^{q}(\{\},\{+a\}))\ ;\ ABs)$, which can only be continued by failing suffixes. However P induces the following computation prefix P' for

$\mathcal{C}((||_{k=1}^{q}(\{\},\{+a\}))\ ;\ (ABs + (\{\},\{+a\})))$ which admits only successful computations:

$$\begin{aligned} P' = \langle \mathcal{C}((||_{k=1}^{q}(\{\},\{+a\}))\ ;\ (ABs + (\{\},\{+a\})))|\varnothing\rangle \\ \rightarrow \cdots \rightarrow \quad \langle \mathcal{C}(ABs + (\{\},\{+a\}))|\ \cup_{k=1}^{q} S_a\rangle \\ \rightarrow \quad \langle B_i|(\cup_{k=1}^{q} S_a)\backslash\{\overline{u_i}\}\rangle \end{aligned}$$

Hence the contradiction.

The fact that $\{u_1,\cdots,u_q\}\cap S'_b = \varnothing$ is proved similarly, by considering S'_b instead of S_a and $(\{\},\{+b\})$ instead of $(\{\},\{+a\})$. □

In order to prove the next proposition, we need to use the lemma 2 of section 3.2.5.

Proposition 88. $\mathcal{L}_{DB}$(nask,get,tell) $\wr$ $\mathcal{L}_{MR}$(nask,tell)

Proof. **(i)** On the one hand, $\mathcal{L}_{DB}$(nask,get,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(nask,tell). Otherwise, by the pattern 2 of transitivity, $\mathcal{L}_{DB}$(ask,tell) $\leqslant$ $\mathcal{L}_{MR}$(nask,tell) which contradicts proposition 81.

(ii) On the other hand, $\mathcal{L}_{MR}$(nask,tell) $\not\leqslant$ $\mathcal{L}_{DB}$(nask,get,tell) is established by contradiction, similarly to the proofs of $\mathcal{L}_{MR}$(nask,tell) $\not\leqslant$ $\mathcal{L}_{DB}$(ask,nask,tell) of proposition 82, which itself extends that of the proof of the second part of proposition 79.

Given the destructive character of get primitives, we shall enrich them with the saturation technique of the proof of the second part of proposition 87 which technically leads to considering the set S'_b instead of the set S_b defined in the second part of the proof of proposition 79. Using these notations and following proof technique 4, we thus fix a token a and reason on two cases, both leading to a contradiction: (I) either there exists a token b such that $S_a \cap S'_b = \varnothing$, (II) or, for any token b, one has $S_a \cap S'_b \neq \varnothing$.

CASE I: there is a token b such that $S_a \cap S'_b = \varnothing$. Consider then $AB = (\{-a,-b\},\{\})$ and $\mathcal{C}(AB)$ in its normal form:

$$\begin{aligned} & tell(\overline{t_1})\ ;\ A_1 + \cdots + tell(\overline{t_p})\ ;\ A_p \\ + \quad & get(\overline{u_1})\ ;\ B_1 + \cdots + get(\overline{u_q})\ ;\ B_q \\ + \quad & nask(\overline{v_1})\ ;\ C_1 + \cdots + nask(\overline{v_r})\ ;\ C_r \end{aligned}$$

As in the proof of the second part of proposition 79, it is possible to establish that there are no alternatives guarded by a $tell(\overline{t_i})$ primitive : if this was the case then, by posing $A = (\{\},\{+a\})$, the agent AB would point out a deadlock for $A\ ;\ (AB + A)$ which only admits successful computations. As in the proof of the second part of proposition 79 also, it is possible to establish that the v_i's should belong to S_a and to S'_b, which amounts to stating that there are no alternatives guarded by a $nask(\overline{v_j})$ primitive.

Consequently, $\mathcal{C}(AB)$ rewrites as

$$get(\overline{u_1})\ ;\ B_1 + \cdots + get(\overline{u_q})\ ;\ B_q$$

and thus $\mathcal{O}(\mathcal{C}(AB)) = \{(\varnothing, \delta^-)\}$ which, by P_3, contradicts the fact that $\mathcal{O}(AB) = \{(\varnothing, \delta^+)\}$.

CASE II: for any token b, one has $S_a \cap S'_b \neq \varnothing$. By Lemma 2 (where S_a plays the role of S and f is defined by $f(x) = S'_x$), there exists a denumerable set of distinct tokens x_i, also distinct from a, and an integer m, such that $\cap_{i=1}^{m}(S_a \cap S'_{x_i}) \neq \varnothing$ and $[\cap_{i=1}^{m}(S_a \cap S'_{x_i})] \cap (S_a \cap S'_{x_j}) \neq \varnothing$, for $j > m$.

Consider $NT = (\{-a, -x_1, \cdots, -x_m\}, \{\})$ and $\mathcal{C}(NT)$ in the following normal form:

$$\begin{array}{l} tell(\overline{t_1})\ ;\ A_1 + \cdots + tell(\overline{t_p})\ ;\ A_p \\ + get(\overline{u_1})\ ;\ B_1 + \cdots + get(\overline{u_q})\ ;\ B_q \\ + nask(\overline{v_1})\ ;\ C_1 + \cdots + nask(\overline{v_r})\ ;\ C_r \end{array}$$

As for case I, it is possible to prove that there are no alternatives guarded by a $tell(\overline{t_i})$ primitive. It is also possible to establish that

$$\{v_1, \cdots, v_r\} \subseteq S_a \cap S'_{x_1} \cap \cdots \cap S'_{x_m}$$

Firstly, we have that $v_k \in S_a$, for any k. Otherwise, assume $v_k \notin S_a$, for some k. Then

$$\begin{array}{rl} F = & \langle \mathcal{C}((\{\}, \{+a\}))\ ;\ \mathcal{C}(NT) \mid \varnothing \rangle \longrightarrow \cdots \\ & \longrightarrow \langle \mathcal{C}(NT) \mid S_a \rangle \longrightarrow \langle C_k \mid S_a \rangle \end{array}$$

would be a valid computation prefix for $\mathcal{C}((\{\}, \{+a\})\ ;\ NT)$ which, by property P_3, can only be continued by failing suffixes. However F induces the following computation prefix F' for $\mathcal{C}((\{\}, \{+a\})\ ;\ (NT + (\{\}, \{+a\})))$, and thus a failing computation for it, which by P_3 contradicts the fact that $(\{\}, \{+a\})\ ;\ (NT + (\{\}, \{+a\})))$ has only one successful computation.

Secondly, we have that $v_k \in S'_{x_i}$, for any k and i. By contradiction, assume that $v_k \notin S'_{x_i}$, for some k and i. The proof proceeds similarly by considering $(PP\ ;\ NT)$ instead of $(\{\}, \{+a\})\ ;\ NT)$ and $PP\ ;\ (NT + (\{\}, \{+x_i\}))$ instead of $(\{\}, \{+a\})\ ;\ (NT + (\{\}, \{+a\}))$ with PP being defined as the parallel composition of $n + 2$ occurrences of $(\{\}, \{+x_i\})$ followed by $(\{\}, \{+a\})$. To that end, note that the computation of $\mathcal{C}(PP)$ leads to the store S'_{x_i} (see the proof of the second part of proposition 83).

Consider now $(\{\}, \{+x_{m+1}\})\ ;\ NT$. A possible computation prefix for $\mathcal{C}((\{\}, \{+x_{m+1}\})\ ;\ NT)$ is, by P_2, as follows:

$$\langle \mathcal{C}((\{\}, \{+x_{m+1}\}))\ ;\ \mathcal{C}(NT) \mid \varnothing \rangle \longrightarrow^* \langle \mathcal{C}(NT) \mid S_{x_{m+1}} \rangle$$

Since $(\{\}, \{+x_{m+1}\})\ ;\ NT$ has a successful computation, and since $\{v_1, \cdots, v_r\} \subseteq S_a \cap S_{x_1} \cap \cdots \cap S_{x_m} \subseteq S_{x_{m+1}}$ there should exist j such that $u_j \in S_{x_{m+1}}$.

Therefore, as $S_{x_{m+1}} \subseteq S'_{x_{m+1}}$, the following derivation is valid:

$$\begin{aligned} H = \ & \langle \mathcal{C}((\ \|_{k=1}^{n+2}(\{\}, \{+x_{m+1}\}))) \ ; \ \mathcal{C}((\{\}, \{+a\})) \ ; \ \mathcal{C}(NT) \mid \varnothing \rangle \\ & \longrightarrow^* \langle \mathcal{C}(NT) \mid S'_{x_{m+1}} \rangle \\ & \longrightarrow \langle B_j \mid S'_{x_{m+1}} \backslash \{\overline{u_j}\} \rangle \end{aligned}$$

Moreover, H should be continued by failing suffixes only since $(\ \|_{k=1}^{n+2}(\{\}, \{+x_{m+1}\})) \ ; \ (\{\}, \{+a\}) \ ; \ NT$ fails. However, by P_3, this introduces failing computations for $(\ \|_{k=1}^{n+2}(\{\}, \{+x_{m+1}\})) \ ; \ (\{\}, \{+a\}) \ ; \ (NT + (\{\}, \{+a\}))$ whereas this agent has only one successful computation. □

We have that $\mathcal{L}_{DB}$(get,tell) is not comparable with $\mathcal{L}_{MR}$(nask,tell).

Proposition 89. $\mathcal{L}_{DB}$(get,tell) $\wr$ $\mathcal{L}_{MR}$(nask,tell)

Proof. On the one hand, $\mathcal{L}_{DB}$(get,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(nask,tell). Otherwise, by pattern 2, as $\mathcal{L}_{DB}$(ask,tell) $<$ $\mathcal{L}_{DB}$(get,tell) (see proposition 66), one would have $\mathcal{L}_{DB}$(ask,tell) $\leqslant$ $\mathcal{L}_{DB}$(get,tell) $\leqslant$ $\mathcal{L}_{MR}$(nask,tell) which has been proved impossible in proposition 81.

On the other hand, $\mathcal{L}_{MR}$(nask,tell) $\not\leqslant$ $\mathcal{L}_{DB}$(get,tell). Otherwise, by pattern 2, we would have $\mathcal{L}_{MR}$(nask,tell) $\leqslant$ $\mathcal{L}_{DB}$(get,tell) $\leqslant$ $\mathcal{L}_{DB}$(nask,get,tell) which has been proved impossible in proposition 88. □

$\mathcal{L}_{DB}$(nask,get,tell) is not comparable with $\mathcal{L}_{MR}$(ask,nask,tell).

Proposition 90. $\mathcal{L}_{MR}$(ask,nask,tell) $\wr$ $\mathcal{L}_{DB}$(nask,get,tell)

Proof. **(i)** Otherwise, $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_{DB}$(nask,get,tell) which contradicts proposition 87. **(ii)** By contradiction, consider $tell(t(1)) \ ; \ get(t(1))$. $\mathcal{O}((tell(t(1)) \ ; \ get(t(1))) = \{(\varnothing, \delta^+)\}$. Hence any computation of $\mathcal{C}(tell(t(1))) \ ; \ \mathcal{C}(get(t(1)))$ is successful. Such a computation is composed of a computation for $\mathcal{C}(tell(t(1)))$ followed by a computation for $\mathcal{C}(get(t(1)))$. As $\mathcal{C}(get(t(1)))$ is composed of ask, nask, tell primitives which do not destroy elements on the store, the latter computation can be repeated step by step which yields successful computation for $\mathcal{C}(tell(t(1))) \ ; \ (\mathcal{C}(get(t(1))) \ \| \ \mathcal{C}(get(t(1))))$. However, $\mathcal{O}(tell(t(1)) \ ; \ (get(t(1)) \ \| \ get(t(1))) = \{(\varnothing, \delta^-)\}$. □

Figure 6.4 presents the expressive relations established up to now, with only considering the three primitives *tell*, *ask* and *nask* in the multi-set rewriting language MRT.

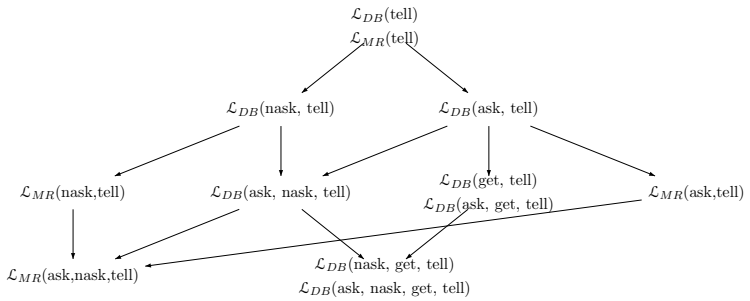


Figure 6.4: Embedding hierarchy of Dense Bach and a multi-set rewriting language, considering the presence of the *tell*, *ask* and *nask* primitives in the mutli-set rewriting language.

6.2.4 Retrieving tokens from the store

This section studies presents the expressiveness relations when introducing the *get* primitive in the sublanguages of MRT. The first proposition establishes that $\mathcal{L}_{DB}$(get,tell) is strictly less expressive than $\mathcal{L}_{MR}$(get,tell).

Proposition 91. $\mathcal{L}_{DB}(\text{get,tell}) < \mathcal{L}_{MR}(\text{get,tell})$

Proof. **(i)** On the one hand, $\mathcal{L}_{DB}(\text{get,tell}) \leqslant \mathcal{L}_{MR}(\text{get,tell})$ holds by proposition 76.

(ii) On the other hand, $\mathcal{L}_{MR}(\text{get,tell}) \nleqslant \mathcal{L}_{DB}(\text{get,tell})$ may be proved as for $\mathcal{L}_{MR}(\text{ask,tell}) \nleqslant \mathcal{L}_{DB}(\text{ask,tell})$ in the proof of the second part of proposition 78. This amounts to considering some of the $ask(u_i)$ to be $get(u_i)$ but does not affect the proof further. Intuitively, $\mathcal{L}_{DB}$(get,tell) is unable to atomically retrieve a and b. Let us thus consider $AB = (\{+a, +b\}, \{-a, -b\})$ and assume that $\mathcal{C}(AB)$ is in normal form and thus is written as

$$tell(\overline{t_1}); A_1 + \cdots + tell(\overline{t_p}); A_p + get(\overline{u_1}); B_1 + \cdots + get(\overline{u_q}); B_q$$

where $\overline{t_i}$ and $\overline{u_j}$ denote the token t_i and u_j associated with a density.

The proof proceeds as explained in proof technique 3 by establishing (I) that there is no alternative guarded by a $tell(\overline{t_i})$ operation, and (II) that there is no alternative guarded by a $get(\overline{u_j})$ operation, in which case, $\mathcal{C}(AB)$ is equivalent to an empty statement, which is not possible since it should contain at least one primitive.

STEP I: Let us first establish that there is no existence of an alternative guarded by a $tell(\overline{t_i})$ operation. Otherwise it would point out a failing computation for $\mathcal{C}(AB + (\{\}, \{+a\}))$, contradicting the fact that $\mathcal{O}(AB + (\{\}, \{+a\})) = (\{a\}, \delta^+)$.

STEP II: Let us now establish that there is no alternative guarded by a $get(\overline{u_j})$ operation. To that end, let us first consider two auxiliary computations: as $\mathcal{O}((\{\}, \{+a\})) = \{(\{a\}, \delta^+)\}$, any

computation of $\mathcal{C}((\{\},\{+a\}))$ starting in the empty store succeeds. Let $\langle((\{\},\{+a\}))|\varnothing\rangle \rightarrow \cdots \rightarrow \langle E|\{a_1,\ldots,a_m\}\rangle$ be such a computation. Similarly, let $\langle((\{\},\{+b\}))|\varnothing\rangle \rightarrow \cdots \rightarrow \langle E|\{b_1,\ldots,b_n\}\rangle$ be one computation of $\mathcal{C}((\{\},\{+b\}))$. The proof of the claim proceeds by establishing, as for proposition 78, that none of the u_i's belong to $\{a_1,\ldots,a_m\} \cup \{b_1,\ldots,b_n\}$, in which case a contradiction occurs from the analysis of $\mathcal{C}((\{\},\{+a\});(\{\},\{+b\});AB)$. As a result, none of the u_i's exist, namely there is no alternative guarded by a $get(\overline{u_j})$ operation. □

We can now prove that $\mathcal{L}_{MR}$(get,tell) is not comparable with respect to $\mathcal{L}_{DB}$(nask,tell), $\mathcal{L}_{DB}$(nask,get,tell) and $\mathcal{L}_{DB}$(ask,nask,tell).

Proposition 92. $\mathcal{L}_{MR}$(get,tell) $\wr$ $\mathcal{L}_{DB}$(nask,tell)

Proof. On the one hand, $\mathcal{L}_{MR}$(get,tell) $\not\leqslant$ $\mathcal{L}_{DB}$(nask,tell). Otherwise, by pattern 2 of transitivity, $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_{MR}$(nask,tell) which has been proved impossible in [BJ03b].

On the other hand, $\mathcal{L}_{DB}$(nask,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(get,tell) is established by contradiction, by considering $tell(t(1))\,;\,nask(t(1))$. Indeed, one has $\mathcal{O}(tell(t(1))\,;\,nask(t(1))) = \{(\{t(1)\},\delta^-)\}$ whereas it is possible to establish that $\mathcal{C}(tell(t(1)))\,;\,\mathcal{C}(nask(t(1)))$ has a successful computation. This is proved by using a reasoning similar to the one used for the second part of proposition 67. □

Proposition 93. $\mathcal{L}_{MR}$(get,tell) $\wr$ $\mathcal{L}_{DB}$(nask,get,tell)

Proof. On the one hand, $\mathcal{L}_{MR}$(get,tell) $\not\leqslant$ $\mathcal{L}_{DB}$(nask,get,tell). Otherwise, by pattern 2 of transitivity, as $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_{MR}$(get,tell), we then have $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_{DB}$(nask,get,tell) which has been proved impossible in proposition 87. On the other hand, $\mathcal{L}_{DB}$(nask,get,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(get,tell). Otherwise, by pattern 2, we would have $\mathcal{L}_{DB}$(nask,tell) $\leqslant$ $\mathcal{L}_{MR}$(get,tell) which has been proved impossible in proposition 92. □

Proposition 94. $\mathcal{L}_{MR}$(get,tell) $\wr$ $\mathcal{L}_{DB}$(ask,nask,tell)

Proof. On the one hand, $\mathcal{L}_{MR}$(get,tell) $\not\leqslant$ $\mathcal{L}_{DB}$(ask,nask,tell). Otherwise, by pattern 2, one has $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_{MR}$(get,tell) $\leqslant$ $\mathcal{L}_{DB}$(ask,nask,tell) which has been proved impossible in proposition 83.

On the other hand, $\mathcal{L}_{DB}$(ask,nask,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(get,tell). Otherwise, by pattern 2, one would have $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_{MR}$(ask,nask,tell) $\leqslant$ $\mathcal{L}_{DB}$(get,tell) which has been proved impossible in proposition 85. □

Figure 6.5 complements Figure 6.4 with the introduction of the *get* primitive inside the subset of the multi-set rewriting language, relating them with the languages of the Dense Bach

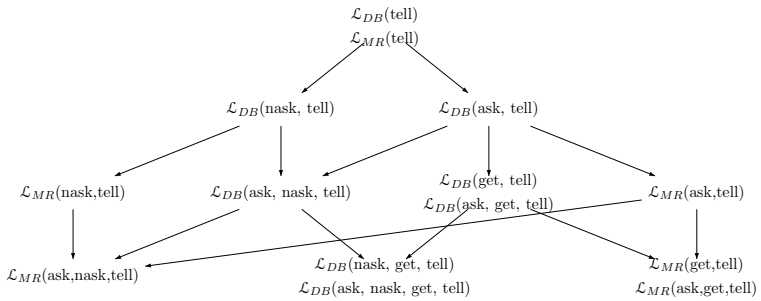


Figure 6.5: Embedding hierarchy of Dense Bach and a multi-set rewriting language, considering the presence of the *get* primitive in the mutli-set rewriting language.

hierarchy.

6.2.5 Checking for the presence and/or absence when adding and/or retrieving tokens

We now prove that $\mathcal{L}_{DB}$(ask,nask,get,tell) is strictly less expressive than $\mathcal{L}_{MR}$(ask,nask,get,tell).

Proposition 95. $\mathcal{L}_{DB}$(ask,nask,get,tell) $< \mathcal{L}_{MR}$(ask,nask,get,tell)

Proof. **(i)** On the one hand, $\mathcal{L}_{DB}$(ask,nask,get,tell) $\leqslant \mathcal{L}_{MR}$(ask,nask,get,tell) is immediate by proposition 76. **(ii)** On the other hand, $\mathcal{L}_{MR}$(ask,nask,get,tell) $\nleqslant \mathcal{L}_{DB}$(ask,nask,get,tell) is established by contradiction, using pattern 2 of transitivity. Indeed, assuming that $\mathcal{L}_{MR}$(ask,nask,get,tell) $\leqslant \mathcal{L}_{DB}$(ask,nask,get,tell), as $\mathcal{L}_{DB}$(ask,nask,get,tell) $=$ $\mathcal{L}_{DB}$(nask,get,tell), one would have $\mathcal{L}_{MR}$(nask, tell) $\leqslant \mathcal{L}_{MR}$(ask,nask,get,tell) $\leqslant$ $\mathcal{L}_{DB}$(ask,nask,get,tell) $\leqslant \mathcal{L}_{DB}$(nask,get,tell) which has been proved impossible in proposition 88. □

Figure 6.6 presents the most complete view of all the expressiveness relations between the different sublanguages of Dense Bach and of the multi-set rewriting language. As for the relation between BachT and Dense Bach, it is worth observing that apart from $\mathcal{L}_{DB}$(tell) $= \mathcal{L}_{MR}$(tell), any sublanguage of Dense Bach is strictly less expressive than the corresponding sublanguage of MRT. Moreover, the very nature of the *tell*, *ask*, *nask* and *get* primitives is kept by MRT, which leads MRT to share the sublanguage hierarchy of the sublanguages of Dense Bach.

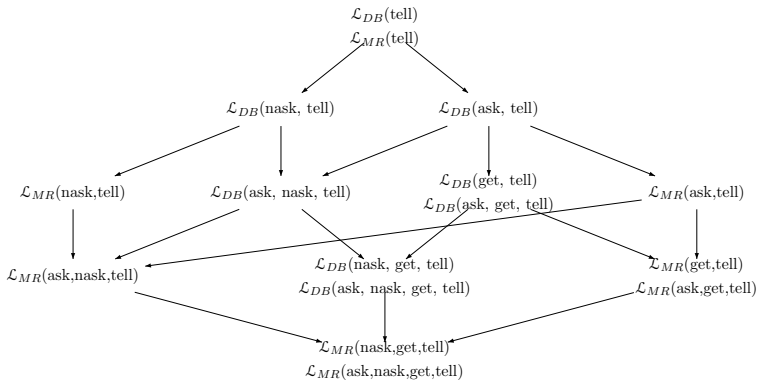


Figure 6.6: Embedding hierarchy of Dense Bach and a multi-set rewriting language, considering the presence of all the primitives in the mutli-set rewriting language.

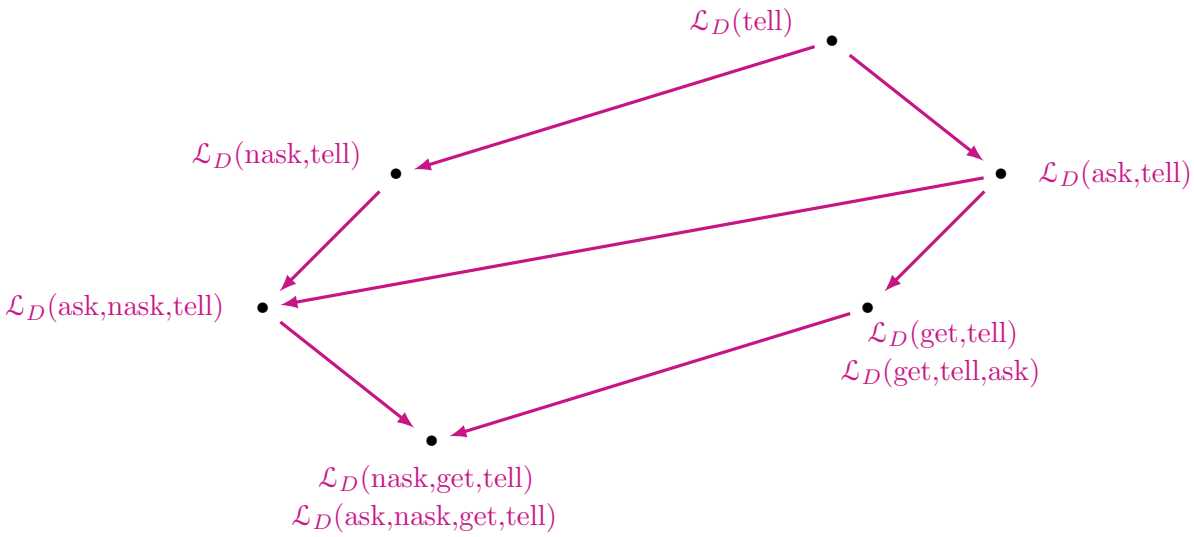


Figure 6.7: Three-dimensional representation of the expressiveness relations between the different sublanguages of Dense Bach.

Figure 6.7 shows a 3-dimension view of the expressiveness relations between the different Dense Bach sublanguages. Figure 6.8 sums up all the relations obtained in this chapter by combining Figures 6.3 and 6.6. Again, to ease the reading, it is drawn in a 3-dimension flavour.

6.3 Conclusion

This chapter has developped a full study of the expressiveness relations between the different sublanguages of Dense Bach and with respect to the BachT and MRT languages. For both studies, the different proposals have been grouped by following a logical approach consisting first in considering the feeding of the shared store with the *tell* primitive, secondly the questioning

of the same store about the presence and or absence of tokens on it, thirdly the retrieval of tokens through the *get* primitive, and finally the consideration of the most complete languages, regrouping all the primitives.

Apart for the *tell* primitive with which BachT, Dense Bach and MRT match, it results from both studies that Dense Bach is strictly more expressive than BachT, but is strictly less expressive than the multi-set rewriting language MRT. In the last case, this results from the possibility for the multi-set rewriting language not only to atomically manipulate many instances of a same token, just as Dense Bach does, but also to manipulate at the same time different tokens. This possibility is clearly out of the scope of the Dense Bach definition language.

The introduction of the density does not alter the very nature of the *tell*, *ask*, *get* and *nask* primitives. As a consequence, the hierarchies of the expressiveness relations between the different sublanguages is similar for BachT, Dense Bach and MRT. This is summarized in the tri-dimensional Figure 6.8.

Our expressiveness studies have been made on the basis of a fixed operational semantics, in contrast with some other work [dBP94] considering different semantics. Moreover the comparison criteria used in [Zav98a, Zav98b] are different from ours, as it is performed on the compositionality of the encoding with respect to parallel composition, the preservaion of parallel and deadlock and a symmetry condition.

A tabulated result of the expressiveness studies between the different sublanguages of BachT, Dense Bach and MRT is presented in Table 6.1. All the possible sublanguages are written in line as well as in column of this table. The intersection indicates for every pair their expressiveness relation, as well as a reference to the proof. A number points to the proof developed in the thesis and a reference indicates a publication from other authors.

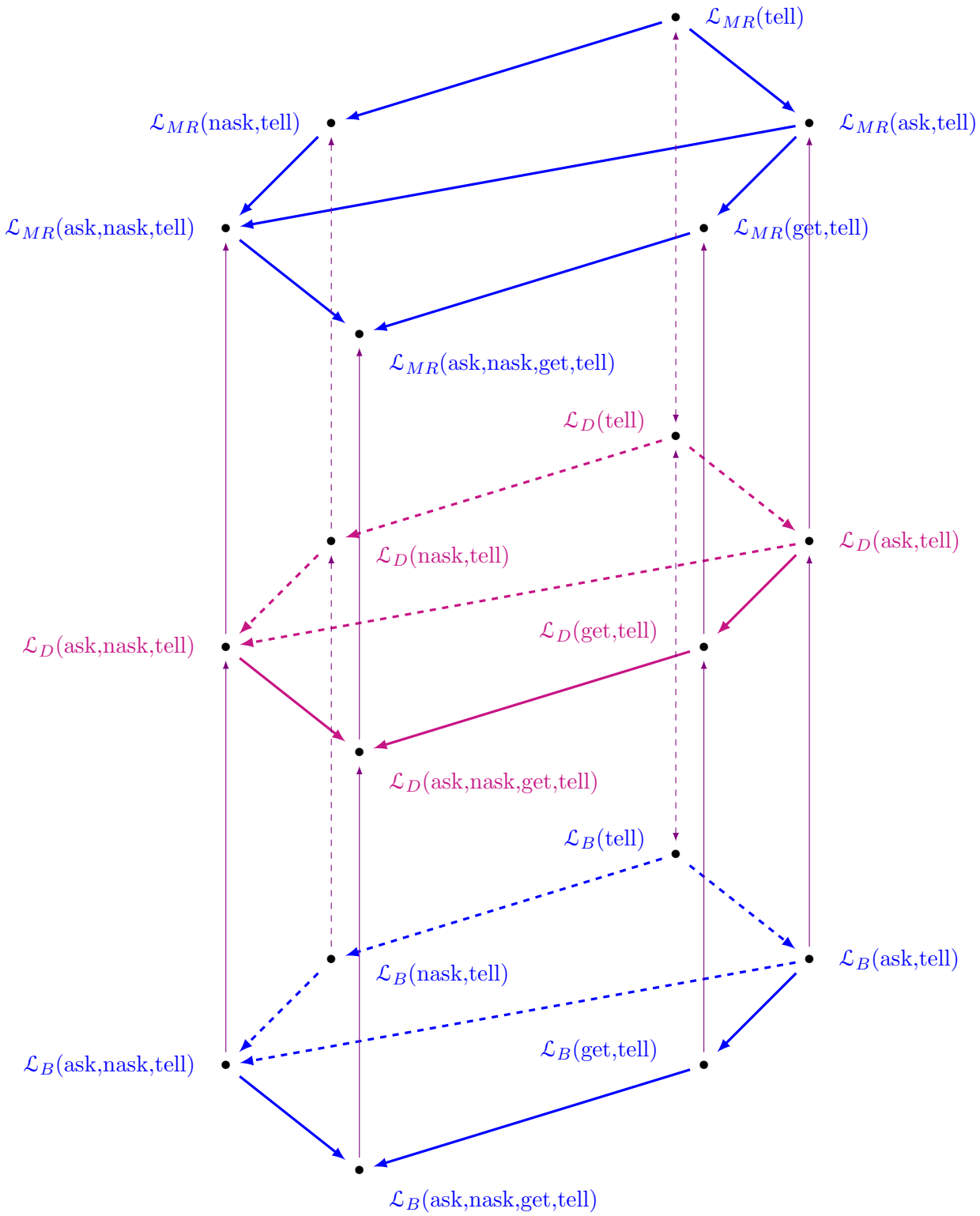


Figure 6.8: Three-dimensional representation of the expressiveness relations between the different sublanguages of BachT, Dense Bach and MRT.

Put reduced figure here	$\mathcal{L}_B$(tell), $\mathcal{L}_{DB}$(tell), $\mathcal{L}_{MR}$(tell)	$\mathcal{L}_B$(ask,tell)	$\mathcal{L}_B$(nask,tell)	$\mathcal{L}_B$(get,tell), $\mathcal{L}_B$(ask,get,tell)	$\mathcal{L}_B$(ask,nask,tell)	$\mathcal{L}_B$(nask,get,tell), $\mathcal{L}_B$(ask,nask,get,tell)	$\mathcal{L}_{DB}$(ask,tell)	$\mathcal{L}_{DB}$(nask,tell)	$\mathcal{L}_{DB}$(get,tell), $\mathcal{L}_{DB}$(ask,get,tell)	$\mathcal{L}_{DB}$(ask,nask,tell)	$\mathcal{L}_{DB}$(nask,get,tell), $\mathcal{L}_{DB}$(ask,nask,get,tell)	$\mathcal{L}_{MR}$(ask,tell)	$\mathcal{L}_{MR}$(nask,tell)	$\mathcal{L}_{MR}$(get,tell), $\mathcal{L}_{MR}$(ask,get,tell)	$\mathcal{L}_{MR}$(ask,nask,tell)	$\mathcal{L}_{MR}$(nask,get,tell), $\mathcal{L}_{MR}$(ask,nask,get,tell)
$\mathcal{L}_B$(tell), $\mathcal{L}_{DB}$(tell), $\mathcal{L}_{MR}$(tell)	= [BJ03b], 37, 77	< [BJ98]	< [BJ98]	< 2	< 2	< 2	< 2	< 2	< 2	< 2	< 2	< 2	< 2	< 2	< 2	< 2
$\mathcal{L}_B$(ask,tell)		=	≀ [BJ98]	< [BJ98]	< [BJ98]	< 2	< 38	≀ 42	< 2	< 2	< 2	< [BJ03b]	≀ [BJ03b]	< 2	< 2	< 2
$\mathcal{L}_B$(nask,tell)			=	≀ [BJ98]	< [BJ98]	< 2	≀ 43	< 40	≀ 67	< 2	< 2	≀ [BJ03b]	< [BJ03b]	≀ [BJ03b]	< [BJ03b]	< [BJ03b]
$\mathcal{L}_B$(get,tell), $\mathcal{L}_B$(ask,get,tell)				= [BJ98]	≀ [BJ98]	< [BJ98]	≀ 46	≀ 59	< 65	≀ 51	< 2	≀ [BJ03b]	≀ [BJ03b]	< [BJ03b]	≀ [BJ03b]	< 2
$\mathcal{L}_B$(ask,nask,tell)					=	< [BJ98]	≀ 52	≀ 45	≀ 71	< 56	< 2	≀ [BJ03b]	≀ [BJ03b]	≀ [BJ03b]	< [BJ03b]	< 2
$\mathcal{L}_B$(nask,get,tell), $\mathcal{L}_B$(ask,nask,get,tell)						= [BJ98]	≀ 53	≀ 57	≀ 69	≀ 58	< 75	≀ [BJ03b]	≀ [BJ03b]	≀ [BJ03b]	≀ [BJ03b]	< [BJ03b]
$\mathcal{L}_{DB}$(ask,tell)							=	≀ 44	< 66	< 56	< 2	< 78	≀ 81	< 2	< 2	< 2
$\mathcal{L}_{DB}$(nask,tell)								=	≀ 68	< 54	< 2	≀ 80	< 79	≀ 92	< 2	< 2
$\mathcal{L}_{DB}$(get,tell), $\mathcal{L}_{DB}$(ask,get,tell)									= 64	≀ 70	< 74	≀ 85	≀ 89	< 91	≀ 86	< 2
$\mathcal{L}_{DB}$(ask,nask,tell)										=	< 73	≀ 83	≀ 82	≀ 94	< 84	< 2
$\mathcal{L}_{DB}$(nask,get,tell), $\mathcal{L}_{DB}$(ask,nask,get,tell)											= 72	≀ 87	≀ 88	≀ 93	≀ 90	< 95
$\mathcal{L}_{MR}$(ask,tell)												=	≀ [BJ03b]	< [BJ03b]	< [BJ03b]	< [BJ03b]
$\mathcal{L}_{MR}$(nask,tell)													=	≀ [BJ03b]	< [BJ03b]	< [BJ03b]
$\mathcal{L}_{MR}$(get,tell), $\mathcal{L}_{MR}$(ask,get,tell)														=	≀ [BJ03b]	< [BJ03b]
$\mathcal{L}_{MR}$(ask,nask,tell)															=	< [BJ03b]
$\mathcal{L}_{MR}$(nask,get,tell), $\mathcal{L}_{MR}$(ask,nask,get,tell)																=

Table 6.1: Table summarizing the expressiveness comparisons between the different sublanguages of BachT, Dense Bach and MRT.

Chapter 7

Expressiveness Study of Vectorized Dense Bach

This chapter positions the Vectorized Dense Bach language from an expressiveness point of view. The first section compares it with Dense Bach. The second section compares it with MRT.

As for the expressiveness studies conducted before in the thesis, we shall use to that end the modular embedding technique proposed by De Boer and Palamidessi (see [dBP94]), in the form slightly redefined in section 3.2.1, and already employed in sections 3.2.4 and 3.2.5.

In both case, we shall also restrict to the relevant sublanguages, namely those that embody the *tell* primitive.

7.1 Comparison with Dense Bach

7.1.1 Generic patterns and results

It is first worth observing that the generic patterns introduced in section 3.2.2 also applies in the context of Vectorized Dense Bach. As a reminder, they embody the following reasonings. The first pattern, named the pattern of sublanguage inclusion, establishes that any language embeds its sublanguages. The second pattern, named pattern of transitivity, takes advantage of transition to establish that $\mathcal{L}_1 \leqslant \mathcal{L}_3$ from the facts that $\mathcal{L}_1 \leqslant \mathcal{L}_2$ and $\mathcal{L}_2 \leqslant \mathcal{L}_3$. The third pattern relies on the contraposition of the second pattern, to establish the non existence of the embedding of $\mathcal{L}_2$ in $\mathcal{L}_3$, namely $\mathcal{L}_2 \not\leqslant \mathcal{L}_3$, from the facts that $\mathcal{L}_1 \leqslant \mathcal{L}_2$ and $\mathcal{L}_1 \not\leqslant \mathcal{L}_3$.

For what concerns the Vector Dense Bach language alone, it is clear by the first pattern

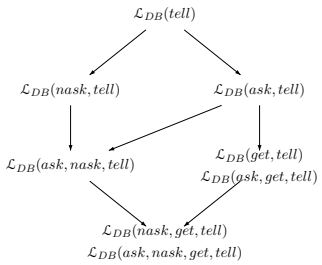


Figure 7.1: Embedding hierarchy of Dense Bach Languages.

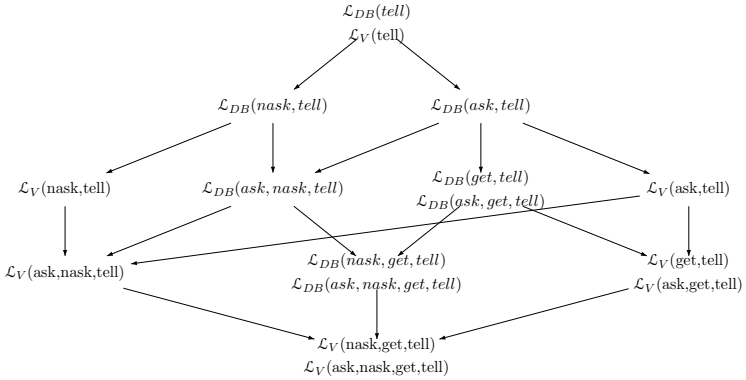


Figure 7.2: Integrated hierarchies of Dense Bach and Dense Bach with Distributed Density.

(pattern 1 of sublanguage inclusion) that a number of modular embeddings are directly established. This first property is formally expressed into the following proposition.

Proposition 96. $\mathcal{L}_V(\ \psi) \leqslant \mathcal{L}_V(\ \chi)$, for any subsets of ψ, χ of primitives such that $\psi \subseteq \chi$.

As a second result, it is also clear that a simple relation exists between the Dense Bach primitives and their equivalent form in Vectorized Dense Bach, simply by taking a list containing only one token. As a result, Dense Bach sublanguages are embedded in the corresponding Vectorized Dense Bach sublanguages, as expressed into the following proposition.

Proposition 97. $\mathcal{L}_{DB}(\ \chi) \leqslant \mathcal{L}_V(\ \chi)$, for any subset of χ of primitives.

Proof. Immediate by defining the coder as follows:

$$\begin{array}{rclcrcl} \mathcal{C}(tell(t(m))) & = & tell(\overrightarrow{t(m)}) & & \mathcal{C}(get(t(m))) & = & get(\overrightarrow{t(m)}) \\ \mathcal{C}(ask(t(m))) & = & ask(\overrightarrow{t(m)}) & & \mathcal{C}(nask(t(m))) & = & nask(\overrightarrow{t(m)}) \end{array}$$

□

More deeply this expresses that Dense Bach is a special case of Vectorized Dense Bach. As the introduction of a list of tokens cannot alter the very nature of the four primitives *tell*,

ask, *nask* and *get*, a similarity between the respective hierarchies of Vectorized Dense Bach sublanguages and of Dense Bach sublanguages, is to be expected, as showned in Figure 3.5 of Section 3.2.2. Nevertheless, even if the global expressive behaviour stays the same, it is also expected that the complete picture of the two hierarchies will reflect an increase in expressiveness in favour of Vectorized Dense Bach with regard to Dense Bach. Indeed, apart for the *tell* primitive, the introduction of a list of tokens gives to *ask*, *nask* and *get* a more efficient atomic behaviour than for their Dense Bach corresponding primitives.

After those general and intuitive results, the next subsections present the detailed proofs of the different embedding relations existing, on the one hand, between all the sublanguages of Vectorized Dense Bach and, on the other hand, between the respective sublanguages of Dense Bach and Vectorized Dense Bach. We shall proceed to the end according to the logical introduction of primitives already used for studying the expressiveness of BachT and MRT. As a result, we first consider placing (by *tell*) some tokens on the store. Then we allow the sublanguages to question the state of the store by introducing *ask* and *nask* primitives. We thereafter allow to retrieve (by means of *get* primitives) some tokens. Finally, we study the combination of the removal of tokens with the check for their presence and/or absence.

7.1.2 Adding tokens on the store

Proposition 98. $\mathcal{L}_{DB}(\text{tell})$ and $\mathcal{L}_{V}(\text{tell})$ are equivalent.

Proof. Indeed, thanks to proposition 97, $\mathcal{L}_{DB}(\text{tell}) \leqslant \mathcal{L}_{V}(\text{tell})$. Furthermore, it is possible to translate any $tell(\overrightarrow{t(m)})$ primitive in the following way. Following Definition 13, a vector of dense tokens is a list of dense tokens. Assuming that this list associated to $\overrightarrow{t(m)}$ is equal to $t_1(m_1), \cdots, t_n(m_n)$, then the coding of $tell(\overrightarrow{t(m)})$ can be defined as the sequential composition of n *tell* Dense Bach primitives:

$$\mathcal{C}(tell(\overrightarrow{t(m)})) = tell(t_1(m_1)); \cdots ; tell(t_n(m_n))$$

To complete the embedding, it is sufficient to take the identity function as decoder. □

7.1.3 Checking for presence and/or absence when adding tokens

As a result of the expressiveness hierarchy [DJL13a] (see figure 7.1), it also comes that both languages $\mathcal{L}_{DB}$(ask,tell) and $\mathcal{L}_{DB}$(nask,tell) are strictly more expressive than $\mathcal{L}_{V}(\text{tell})$ since both have been established strictly more expressive than $\mathcal{L}_{DB}(\text{tell})$.

Let us now compare $\mathcal{L}_{DB}$(ask,tell) with its vectorized dense counterpart.

Proposition 99. $\mathcal{L}_{DB}(\text{ask,tell}) < \mathcal{L}_{V}(\text{ask,tell})$

Proof. On the one hand, $\mathcal{L}_{DB}(\text{ask,tell}) \leqslant \mathcal{L}_{V}(\text{ask,tell})$, by proposition 97. On the other hand, $\mathcal{L}_{V}(\text{ask,tell}) \nleqslant \mathcal{L}_{DB}(\text{ask,tell})$ can be established by contradiction. The proof proceeds by exploiting the inability of $\mathcal{L}_{DB}$(ask,tell) to atomically test the presence of two distinct tokens a and b. Assume thus the existence of a coder $\mathcal{C} : \mathcal{L}_{V}(\text{ask,tell}) \to \mathcal{L}_{DB}(\text{ask,tell})$ and consider $AB = ask((a(1), b(1)))$. Let us prove that its coder is empty, which is absurd since, by Definition 22 in section 5.2.2, it should contain at least one primitive. To that end, one may assume that $\mathcal{C}(AB)$ is in normal form [BJ98] and thus is written as $tell(\overline{t_1}); A_1 + \cdots + tell(\overline{t_p}); A_p + ask(\overline{u_1}); B_1 + \cdots + ask(\overline{u_q}); B_q$, where $\overline{t_i}$ and $\overline{u_j}$ denote the token t_i and u_j associated with a density. In this expression, we will establish that there is no alternative guarded by a $tell(\overline{t_i})$ operation, and no alternative guarded by a $ask(\overline{u_j})$ operation either, in which case $\mathcal{C}(AB)$ is empty.

Let us first establish by contradiction that there is no alternative guarded by a $tell(\overline{t_i})$ operation. Assume there is one, say guarded by $tell(\overline{t_i})$. Then $D = \langle \mathcal{C}(AB) | \varnothing \rangle \to \langle A_i | \{\overline{t_i}\} \rangle$ is a valid computation prefix of $\mathcal{C}(AB)$. It should deadlocks afterwards since $\mathcal{O}(AB) = (\varnothing, \delta^-)$. However D is also a valid computation prefix of $\mathcal{C}(AB + tell(a(1)))$. Hence, $\mathcal{C}(AB + tell(a(1)))$ admits a failing computation which contradicts the fact that $\mathcal{O}(AB + tell(a(1))) = (\{a\}, \delta^+)$.

Secondly we establish that there is also no alternative guarded by an $ask(\overline{u_j})$ operation. To that end, let us first consider two auxiliary computations. As $\mathcal{O}(tell(a(1))) = (\{a\}, \delta^+)$, any computation of $\mathcal{C}(tell(a(1)))$ starting in the empty store succeeds. Let $\langle \mathcal{C}(tell(a(1))) | \varnothing \rangle \to \cdots \to \langle E | \{a_1, \ldots, a_m\} \rangle$ be such a computation. Similarly, let $\langle \mathcal{C}(tell(b(1))) | \varnothing \rangle \to \cdots \to \langle E | \{b_1, \ldots, b_n\} \rangle$ be one computation of $\mathcal{C}(tell(b(1))$. The proof of the claim proceeds in two steps. First let us prove that none of the u_i's belong to $\{a_1, \ldots, a_m\}$. By contradiction, assume that $u_i = a_k$ for some k and that d is the density associated with u_i, namely, $\overline{u_i} = u_i(d)$. Let us observe that, since it is in $\mathcal{L}_{DB}$(ask,tell), the considered computation of $\mathcal{C}(tell(a(1))$ can be repeated sequentially, as many times as needed. As a result, by using A^d to denote the sequential composition of d instances of A, the sequence $D' = \langle \mathcal{C}(tell(a(1))^d; AB) | \varnothing \rangle \to \cdots \to \langle \mathcal{C}(AB) | \{a_1^d, \ldots, a_m^d\} \rangle \to \langle B_j | \{a_1^d, \ldots, a_m^d\} \rangle$ is a valid computation prefix of $\mathcal{C}(tell(a(1))^d; AB)$, which can only be continued by failing suffixes. However D' induces the following computation prefix D'' for $tell(a(1))^d; (AB + ask(a(1))$ which admits only successful computations: $D'' = \langle \mathcal{C}(tell(a(1))^d; (AB + ask(a(1)))) | \varnothing \rangle \to \cdots \to \langle \mathcal{C}(AB + ask(a(1))) | \{a_1^d, \ldots, a_m^d\} \rangle \to \langle B_j | \{a_1^d, \ldots, a_m^d\} \rangle$.

The proof proceeds similarly in the case $u_j \in \{b_1, \ldots, b_n\}$ for some $j \in 1, \ldots, q$ by then considering $tell(b(1))^d; AB$ and $tell(b(1))^d; (AB + ask(b(1)))$.

Finally, the fact that the $u_i's$ do not belong to $\{a_1, \ldots, a_m\} \cup \{b_1, \ldots, b_n\}$ induces a

contradiction. Indeed, if this is the case then $\langle \mathcal{C}(tell(a(1)); tell(b(1)); AB) | \varnothing \rangle \rightarrow \cdots \rightarrow \langle \mathcal{C}(tell(b(1)); AB) | \{a_1, \ldots, a_m\} \rangle \rightarrow \cdots \rightarrow \langle \mathcal{C}(AB) | \{a_1, \ldots, a_m, b_1, \ldots, b_n\} \rangle \nrightarrow$ is a valid failing computation prefix of $\mathcal{C}(tell(a(1)); tell(b(1)); AB)$ whereas $tell(a(1)); tell(b(1)); AB$ has only one successful computation. As a conclusion, $\mathcal{C}(AB)$ is equivalent to an empty statement, which is absurd by definition 22. □

Symmetrically, $\mathcal{L}_{DB}$(nask,tell) is strictly less expressive than $\mathcal{L}_V$(nask,tell). To establish this result we shall use again Lemma 1 of Section 3.2.5.

Proposition 100. $\mathcal{L}_{DB}(\text{nask,tell}) < \mathcal{L}_V(\text{nask,tell})$.

Proof. On the one hand, $\mathcal{L}_{DB}(\text{nask,tell}) \leqslant \mathcal{L}_V(\text{nask,tell})$ holds by proposition 97. On the other hand, $\mathcal{L}_V(\text{nask,tell}) \nleqslant \mathcal{L}_{DB}(\text{nask,tell})$ is proved by contradiction, assuming the existence of a coder $\mathcal{C}$. The proof proceeds as in proposition 99 but this time by exploiting the inability of $\mathcal{L}_{DB}$(nask,tell) to atomically test the absence of two distinct tokens a and b. In the following, the construction of the tokens $\{a_1, \ldots, a_m\}$ and $\{b_1, \ldots, b_n\}$ associated with the coding of a and b will be generalized by the definition of a function $f : Stoken \rightarrow \mathcal{P}_f(Stoken)$, associating to each token a finite set of tokens. To that end, as $\mathcal{O}(tell(t(1))) = \{(\{t\}, \delta^+)\}$, for any token t, any computation of $\mathcal{C}(tell(t(1)))$ starting in the empty store succeeds. Let $\langle \mathcal{C}(tell(t(1))) | \varnothing \rangle \rightarrow \cdots \rightarrow \langle E | \{t_1, \ldots, t_{m_t}\} \rangle$ be such a computation and let S_t denote the resulting store $\{t_1, \cdots, t_{m_t}\}$. Then the proof of the claim proceeds by examining two cases: (I) either there exist two tokens a and b such that $S_a \cap S_b = \varnothing$, (II) or $S_a \cap S_b \neq \varnothing$ for any pair of tokens a and b.

CASE I: Let us suppose first that there are two tokens a and b such $S_a \cap S_b = \varnothing$. Consider $AB = nask((a(1), b(1)))$ and $\mathcal{C}(AB)$ in its normal form:

$$tell(\overline{v_1}) \; ; \; A_1 + \cdots + tell(\overline{v_p}) \; ; \; A_p + nask(\overline{u_1}) \; ; \; B_1 + \cdots + nask(\overline{u_q}) \; ; \; B_q$$

The proof then proceeds by establishing that there are no alternatives guarded by $tell(\overline{v_i})$ nor by $nask(\overline{u_j})$. The absence of alternative guarded by a $tell(\overline{v_i})$ primitive is established as in proposition 99(ii): if this was not the case then AB would point out a deadlocking computation for $tell(a(1)); (AB + tell(a(1)))$ which only admits successful computations. To prove the absence of an alternative guarded by a $nask(\overline{u_j})$ primitive, let us establish that the u_j's should belong to S_a and to S_b, which is impossible since $S_a \cap S_b = \varnothing$. By contradiction, assume that $u_j \notin S_a$ for some j (the case where $u_j \notin S_b$ is treated similarly). Then

$$\langle \mathcal{C}(tell(a(1)) \; ; \; AB) \mid \varnothing \rangle \longrightarrow \cdots \longrightarrow \langle \mathcal{C}(AB) \mid S_a \rangle \longrightarrow \langle B_j \mid S_a \rangle$$

is a valid computation prefix of $\mathcal{C}(tell(a(1)) \; ; \; AB)$ which can only be continued by failing suffixes. However, this prefix induces the following computation prefix D' for $\mathcal{C}(tell(a(1)) \; ; \; (AB + tell(a(1))))$ which should only admit successful computations:

$$\langle \mathcal{C}(tell(a(1)) \; ; \; (AB + tell(b(1)))) \mid \varnothing \rangle \longrightarrow \cdots$$

$$\longrightarrow \langle \mathcal{C}(AB + tell(b(1))) \mid S_a \rangle \longrightarrow \langle B_j \mid S_a \rangle$$

CASE II: les us now suppose that $S_a \cap S_b \neq \varnothing$ for any pair of tokens a and b. As proved by Lemma 1, it is possible to construct an infinite sequence of distinct tokens x_i's and to identify an integer n such that

$$\bigcap_{i=1}^{n} S_{x_i} \neq \varnothing$$

and

$$\bigcap_{i=1}^{n} S_{x_i} = \bigcap_{i=1}^{n} S_{x_i} \cap S_{x_j}$$

for any $j > n$. Let us consider now $NT = nask((x_1(n), \cdots, x_n(n)))$ and $\mathcal{C}(NT)$ in its normal form

$$tell(\overline{v_1}) \; ; \; A_1 + \cdots + tell(\overline{v_p}) \; ; \; A_p + nask(\overline{u_1}) \; ; \; B_1 + \cdots + nask(\overline{u_q}) \; ; \; B_q$$

Using similar reasoning as for case I, one may prove that there are no alternatives guarded by a $tell(\overline{v_i})$ primitive and that $\{u_1, \cdots, u_q\} \subseteq S_{x_1} \cap \cdots \cap S_{x_n}$. In this situation, one may hope to conclude that $\mathcal{C}(tell((x_{n+1})(1)) \; ; \; NT)$ has a failing computation since $S_{x_1} \cap \cdots \cap S_{x_n} \cap S_{x_{n+1}} = S_{x_1} \cap \cdots \cap S_{x_n}$ and thus $\{u_1, \cdots, u_q\} \subseteq S_{x_1} \cap \cdots \cap S_{x_n} \subseteq S_{x_{n+1}}$. However, this could actually not be the case because of the multiplicity inside the $nask(\overline{u_j})$ primitives, which allow these primitives to succeed even though u_j is present on the store but in not enough occurences. This may be solved by putting multiple copies of $tell(x_{n+1})$ in parallel. Let α be the addition of the multiplicity of the tokens $\overline{u_1}$, ..., $\overline{u_q}$ and let TT_α denote the parallel composition of α copies of $tell(x_{n+1})$. Then, by repeating in turn the execution of the primitives of $\mathcal{C}(tell((x_{n+1})(1)))$, one reaches the store consisting of α occurrences of $S_{x_{n+1}}$, which allows to conclude that $\mathcal{C}(TT_\alpha \; ; \; NT)$ has a failing computation whereas $TT_\alpha \; ; \; NT$ has only one successful computation.

In conclusion, $\mathcal{C}(AB)$ reduces to an empty statement, which implies that there is no coder $\mathcal{C}$. □

$\mathcal{L}_V$(nask,tell) and $\mathcal{L}_{DB}$(ask,tell) are not comparable with each other.

Proposition 101. $\mathcal{L}_V$(nask,tell) $\wr$ $\mathcal{L}_{DB}$(ask,tell).

Proof. **(i)** $\mathcal{L}_V$(nask,tell) $\nleqslant$ $\mathcal{L}_{DB}$(ask,tell), otherwise by pattern 3 of non embedding by transitivity, we have $\mathcal{L}_{DB}$(nask,tell) $\leqslant$ $\mathcal{L}_{DB}$(ask,tell) which has been proved impossible in proposition 44. **(ii)** $\mathcal{L}_{DB}$(ask,tell) $\nleqslant$ $\mathcal{L}_V$(nask,tell) is proved by contradiction. Consider $A = tell(t(1)) \; ; \; ask(t(1)))$. One has $\mathcal{O}(A) = \{(\{t(1)\}, \delta^+)\}$. Hence, by P_3, $\mathcal{C}(A)$ succeeds whereas we shall establish that it has failing computations. Indeed, since $\mathcal{O}(ask(t(1))) = \{(\varnothing, \delta^-)\}$, any computation of $\mathcal{C}(ask(t(1)))$ starting on the empty store fails. As $\mathcal{C}(ask(t(1)))$ is composed of nask and tell primitives, this can only occur by having a nask primitive preceded by a tell

primitive. As enriching the initial content of the store leads to the same result, any computation starting on any (arbitrary) store fails. As a consequence, even if $\mathcal{C}(tell(t(1)))$ has a successful computation, this computation cannot be continued by a successful computation of $\mathcal{C}(ask(t(1)))$. Consequently any computation of $\mathcal{C}(tell(t(1)); ask(t(1)))$ fails, which produces a contradiction. □

$\mathcal{L}_V$(ask,tell) and $\mathcal{L}_{DB}$(nask,tell) are not comparable with each other.

Proposition 102. $\mathcal{L}_V$(ask,tell) $\wr$ $\mathcal{L}_{DB}$(nask,tell).

Proof. **(i)** $\mathcal{L}_V$(ask,tell) $\nleqslant$ $\mathcal{L}_{DB}$(nask,tell), otherwise by pattern 3 of non embedding by transitivity, we would have $\mathcal{L}_{DB}$(ask,tell) $\leqslant$ $\mathcal{L}_{DB}$(nask,tell) which has been proved impossible in proposition 44. **(ii)** $\mathcal{L}_{DB}$(nask,tell) $\nleqslant$ $\mathcal{L}_V$(ask,tell) is proved by contradiction. Consider $A = tell(t(1))\ ;\ nask(t(1)))$. One has $\mathcal{O}(A) = \{(\{t\}, \delta^-)\}$. By P_3, $\mathcal{C}(A)$ fails, whereas we shall establish that it has a successful computation. Indeed, since $\mathcal{O}(tell(t(1))) = \{(\{t(1)\}, \delta^+)\}$, any computation of $\mathcal{C}(tell(t(1)))$ starting on the empty store is successful. Similarly, it follows from $\mathcal{O}(nask(t(1))) = \{(\varnothing, \delta^+)\}$ that any computation of $\mathcal{C}(nask(t(1)))$ starting on the empty store is successful, and, consequently, is any computation starting from any store, since $\mathcal{C}(nask(t(1)))$ is composed of ask and tell primitives. Summing up, any (successful) computation of $\mathcal{C}(tell(t(1)))$ starting on the empty store can be continued by a (successful) computation of $\mathcal{C}(nask(t(1)))$, which leads to the contradiction. □

$\mathcal{L}_V$(nask,tell) and $\mathcal{L}_V$(ask,tell) are not comparable with each other.

Proposition 103. $\mathcal{L}_V$(nask,tell) $\wr$ $\mathcal{L}_V$(ask,tell).

Proof. **(i)** $\mathcal{L}_V$(nask,tell) $\nleqslant$ $\mathcal{L}_V$(ask,tell), otherwise by pattern 3 of non embedding by transitivity $\mathcal{L}_{DB}$(nask,tell) $\leqslant$ $\mathcal{L}_V$(ask,tell), which contradicts proposition 102. **(ii)** $\mathcal{L}_V$(ask,tell) $\nleqslant$ $\mathcal{L}_V$(nask,tell), otherwise by the same pattern 3, $\mathcal{L}_{DB}$(ask,tell) $\leqslant$ $\mathcal{L}_V$(nask,tell), which contradicts proposition 101. □

$\mathcal{L}_V$(nask,tell) and $\mathcal{L}_{DB}$(ask,nask,tell) are not comparable with each other.

Proposition 104. $\mathcal{L}_V$(nask,tell) $\wr$ $\mathcal{L}_{DB}$(ask,nask,tell).

Proof. **(i)** $\mathcal{L}_{DB}$(ask,nask,tell) $\nleqslant$ $\mathcal{L}_V$(nask,tell), otherwise by pattern 3 of non embedding by transitivity, $\mathcal{L}_{DB}$(ask,tell) $\leqslant$ $\mathcal{L}_V$(nask,tell), which contradicts proposition 101(ii). **(ii)** For

$\mathcal{L}_V$(nask,tell) $\not\leqslant$ $\mathcal{L}_{DB}$(ask,nask,tell), the proof proceeds as in proposition 100(ii). The presence of the ask primitive in $\mathcal{L}_{DB}$ does not modify the reasoning, as it does not destroy elements and so does not modify the state of the store σ. □

Symmetrically, $\mathcal{L}_{DB}$(get,tell) and $\mathcal{L}_V$(ask,tell) are not comparable with each other.

Proposition 105. $\mathcal{L}_{DB}$(get,tell) $\wr$ $\mathcal{L}_V$(ask,tell).

Proof. **(i)** $\mathcal{L}_{DB}$(get,tell) $\not\leqslant$ $\mathcal{L}_V$(ask,tell) is proved by contradiction. Consider $tell(t(1))$; $get(t(1))$. One has $\mathcal{O}(tell(t(1))$; $get(t(1))) = \{(\varnothing, \delta^+)\}$. By P_2 and P_3, any computation of $\mathcal{O}(\mathcal{C}(tell(t(1)))$; $\mathcal{C}(get(t(1))))$ is thus successful. Since $\mathcal{C}(get(t(1)))$ is composed of ask and tell primitives only and since ask and tell primitives do not destroy elements, at least one computation of $\mathcal{O}(\mathcal{C}(tell(t(1)))$; $\mathcal{C}(get(t(1)))$; $\mathcal{C}(get(t(1))))$ is successful. However, $\mathcal{O}(tell(t(1))$; $get(t(1))$; $get(t(1))) = \{(\varnothing, \delta^-)\}$, which provides the contradiction.

(ii) The proof that $\mathcal{L}_V$(ask,tell) $\not\leqslant$ $\mathcal{L}_{DB}$(get,tell) is established by contradiction. Intuitively, $\mathcal{L}_{DB}$(get,tell) is unable to atomically test the presence of a and b. Let us thus consider $AB = ask((a(1), b(1)))$ and prove that its coder has a successful computation. This leads to a contradiction since AB has just one failing computation. To that end, one may assume that $\mathcal{C}(AB)$ is in normal form (see [BJ98]) and thus is written as $tell(\overline{t_1}); A_1 + \cdots + tell(\overline{t_p}); A_p + get(\overline{u_1}); B_1 + \cdots + get(\overline{u_q}); B_q$, where $\overline{t_i}$ and $\overline{u_j}$ denote the token t_i and u_j associated to a density.

The proof proceeds by establishing that (I) there is no alternative guarded by a $tell(\overline{t_i})$ operation, and (II) there is no alternative guarded by a $get(\overline{u_j})$ operation. In which case, $\mathcal{C}(AB)$ is equivalent to an empty statement, which is not possible in view of definition 22.

CASE I: there is no alternative guarded by a $tell(\overline{t_i})$ operation. Otherwise, $D = \langle \mathcal{C}(AB) | \varnothing \rangle \rightarrow \langle A_i | \{\overline{t_i}\} \rangle$ would be a valid computation prefix of $\mathcal{C}(AB)$ which should deadlocks afterwards since $\mathcal{O}(AB) = \{(\varnothing, \delta^-)\}$. However D is also a valid computation prefix of $\mathcal{C}(AB + tell(a(1)))$. Hence, $\mathcal{C}(AB + tell(a(1)))$ admits a failing computation which contradicts the fact that $\mathcal{O}(AB + tell(a(1))) = (\{a\}, \delta^+)$.

CASE II: there is no alternative guarded by a $get(\overline{u_j})$ operation. To that end, let us first consider two auxiliary computations: as $\mathcal{O}(tell(a(1))) = \{(\{a\}, \delta^+)\}$, any computation of $\mathcal{C}(tell(a(1)))$ starting in the empty store succeeds. Let $\langle \mathcal{C}(tell(a(1))) | \varnothing \rangle \rightarrow \cdots \rightarrow \langle E | \{a_1, \ldots, a_m\} \rangle$ be such a computation. Similarly, let $\langle (tell(b(1))) | \varnothing \rangle \rightarrow \cdots \rightarrow \langle E | \{b_1, \ldots, b_n\} \rangle$ be one computation of $\mathcal{C}(tell(b(1)))$. As these two computations start by assuming no token on the store and since $\mathcal{L}_{DB}$(get,tell) does not contain negative tests, it is easy to verify that they

can be put in sequence so as to establish the following computations:

$$\begin{array}{lll}
\langle \mathcal{C}(tell(a(1)); tell(b(1)))|\varnothing\rangle & \rightarrow \cdots \rightarrow & \langle \mathcal{C}(tell(b(1)))|\{a_1,\ldots,a_m\}\rangle \\
& \rightarrow \cdots \rightarrow & \langle E|\{a_1,\ldots,a_m\} \cup \{b_1,\ldots,b_n\}\rangle \\
\langle \mathcal{C}(tell(b(1)); tell(a(1)))|\varnothing\rangle & \rightarrow \cdots \rightarrow & \langle \mathcal{C}(tell(a(1)))|\{b_1,\ldots,b_n\}\rangle \\
& \rightarrow \cdots \rightarrow & \langle E|\{a_1,\ldots,a_m\} \cup \{b_1,\ldots,b_n\}\rangle
\end{array}$$

As $\mathcal{C}(tell(a(1)); tell(b(1)); AB)$ has a successful computation, one of the $get(\overline{u_i})$ succeeds, and, consequently, one has $\{\overline{u_j}\} \subseteq \{a_1,\ldots,a_m\} \cup \{b_1,\ldots,b_n\}$ for some j. Assume $\overline{u_j} = a_k$ for k and let d be the density associated to u_j, namely, $\overline{u_j} = a_k(d)$. Then

$$D' = \langle \mathcal{C}(tell(a(1)); AB)|\varnothing\rangle \rightarrow \cdots \rightarrow \langle \mathcal{C}(AB)|\{a_1,\ldots,a_m\}\rangle \rightarrow \langle B_j|\{a_1,\ldots,a_m\}\backslash\{\overline{u_j}\}\rangle$$

is a valid computation prefix of $\mathcal{C}(tell(a(1)); AB)$. It can only be continued by failing suffixes since $tell(a(1)); AB$ fails. However, this induces the following computation prefix D'' for $\mathcal{C}(tell(a(1)); (AB + ask(a(1))))$ and thus a failing computation whereas $tell(a(1)); (AB + ask(a(1)))$ only admits a successful computation:

$$\begin{array}{lll}
D'' = \langle \mathcal{C}(tell(a(1)); (AB + ask(a(1))))|\varnothing\rangle & \rightarrow \cdots \rightarrow & \langle \mathcal{C}(AB + ask(a(1)))|\{a_1,\ldots,a_m\}\rangle \\
& \rightarrow & \langle B_j|\{a_1,\ldots,a_m\}\backslash\{\overline{u_j}\}\rangle.
\end{array}$$

The proof proceeds similarly in the case $u_j = b_k$ for some k by then considering $tell(b(1)); AB$ and $tell(b(1)); (AB + ask(b(1)))$. □

We now establish that $\mathcal{L}_{DB}$(get,tell) $\wr$ $\mathcal{L}_V$(ask,nask,tell).

Proposition 106. $\mathcal{L}_{DB}$(get,tell) $\wr$ $\mathcal{L}_V$(ask,nask,tell)

Proof. On the one hand, $\mathcal{L}_{DB}$(get,tell) $\not\leqslant$ $\mathcal{L}_V$(ask,nask,tell). This is established by using the reasoning of the first part of the proof of proposition 105 and by replacing the sequential composition of the two $get(t(1))$ primitives by a parallel one, in order to cope with the potential presence of nask primitives. On the other hand, $\mathcal{L}_V$(ask,nask,tell) $\not\leqslant$ $\mathcal{L}_{DB}$(get,tell) is established by contradiction. Otherwise by the pattern 1 of sublanguage inclusion, $\mathcal{L}_V$(ask,tell) $\leqslant$ $\mathcal{L}_V$(ask,nask,tell) and then $\mathcal{L}_V$(ask,tell) $\leqslant$ $\mathcal{L}_{DB}$(get,tell) holds, which contradicts proposition 105. □

$\mathcal{L}_{DB}$(ask,nask,tell) is not comparable with $\mathcal{L}_V$(ask,tell).

Proposition 107. $\mathcal{L}_{DB}$(ask,nask,tell) $\wr$ $\mathcal{L}_V$(ask,tell)

Proof. **(i)** On the one hand, $\mathcal{L}_{DB}$(ask,nask,tell) $\nleqslant \mathcal{L}_{V}$(ask,tell) is established by contradiction. Otherwise by the pattern 1 of sublanguage inclusion, $\mathcal{L}_{DB}$(nask,tell) $\leqslant \mathcal{L}_{DB}$(ask,nask,tell) and then $\mathcal{L}_{DB}$(nask,tell) $\leqslant \mathcal{L}_{V}$(ask,tell) holds, which contradicts proposition 102.

(ii) On the other hand, $\mathcal{L}_{V}$(ask,tell) $\nleqslant \mathcal{L}_{DB}$(ask,nask,tell) is established by contradiction, by assuming the existence of a coder $\mathcal{C}$ from $\mathcal{L}_{V}$(ask,tell) to $\mathcal{L}_{DB}$(ask,nask,tell). Let n be the cumulative occurrences of tokens in *nask* primitives of $\mathcal{C}(tell(a(1)))$.

As $\mathcal{C}(tell(a(1)))$ has only successful computations, let, as in the proof of proposition 100(ii), S_a be the store resulting from one of them. Moreover, as a matter of notation, let the construction $A^{||q}$ denote the parallel composition of q copies of A. As $(tell(b(1)))^{||(n+2)}$; $tell(a(1)))$ succeeds as well, let S'_b denote the store resulting from one successful computation of its coding. Consider now $ABs = ask(a(1), b(n+3))$ and $\mathcal{C}(ABs)$ in its normal form:

$$\begin{array}{l} tell(\overline{t_1})\ ;\ A_1 + \cdots + tell(\overline{t_p})\ ;\ A_p \\ + ask(\overline{u_1})\ ;\ B_1 + \cdots + ask(\overline{u_q})\ ;\ B_q \\ + nask(\overline{v_1})\ ;\ C_1 + \cdots + nask(\overline{v_r})\ ;\ C_r \end{array}$$

We shall establish that: (I) there are no alternatives guarded by $tell(\overline{t_i})$ and $nask(\overline{v_j})$ primitives, and that (II) $\{u_1, \cdots, u_q\} \cap (S_a \cup S'_b) = \varnothing$. Therefore, as computing once more an instance of $\mathcal{C}(tell(b(1))$ in parallel just add copies of tokens already present in $S_a \cup S'_b$, it follows that $\mathcal{C}((tell(b(1)))^{||(n+3)}\ ;\ tell(a(1))\ ;\ ABs)$ fails, which is absurd by P_3, since, by construction, $(tell(b(1)))^{||(n+3)}\ ;\ tell(a(1))\ ;\ ABs$ has one successful computation.

CASE I: There are no alternatives guarded by a *tell* primitive. The proof proceeds by contradiction as in proposition 99(ii). Assume thus the existence of a $tell(\overline{t_i})\ ;\ A_i$ alternative. Then $D = \langle \mathcal{C}(ABs) | \varnothing \rangle \rightarrow \langle A_i | \{\overline{t_i}\} \rangle$ is a valid computation prefix of $\mathcal{C}(ABs)$ which should deadlock afterwards since $\mathcal{O}(ABs) = \{(\varnothing, \delta^-)\}$. However D is also a valid computation prefix of $\mathcal{C}(ABs + tell(a(1)))$. Hence, $\mathcal{C}(ABs + tell(a(1)))$ admits a failing computation which contradicts the fact that $\mathcal{O}(ABs + tell(a(1))) = \{(\{a\}, \delta^+)\}$.

Moreover, there are no alternatives guarded by a *nask* primitive. In a similar way, assume thus the existence of a $nask(\overline{v_i})\ ;\ C_i$ alternative. Then $D' = \langle \mathcal{C}(ABs) | \varnothing \rangle \rightarrow \langle C_i | \varnothing \rangle$ is a valid computation prefix of $\mathcal{C}(ABs)$ which should deadlock afterwards since $\mathcal{O}(ABs) = \{\varnothing, \delta^-\}$. However D' is also a valid computation prefix of $\mathcal{C}(ABs + tell(a(1)))$. Hence, $\mathcal{C}(ABs + tell(a(1)))$ admits a failing computation which contradicts the fact that $\mathcal{O}(ABs + tell(a(1))) = (\{a\}, \delta^+)$.

CASE II: Let us now prove that $\{u_1, \cdots, u_q\} \cap (S_a \cup S'_b) = \varnothing$. This is proved in two steps by establishing that (1) $\{u_1, \cdots, u_q\} \cap S_a = \varnothing$, and that (2) $\{u_1, \cdots, u_q\} \cap S'_b = \varnothing$.

First let us prove that $\{u_1, \cdots, u_q\} \cap S_a = \varnothing$. Assume $u_i \in S_a$ and let d be the density associated to u_i, namely, $\overline{u_i} = u_i(d)$. Let us observe that each step of the considered computation

of $\mathcal{C}(tell(a(1)))$ can be repeated in turn, in as many parallel occurences of it as needed, so that

$$\begin{aligned} P = \langle \mathcal{C}(tell(a(1))^{||d} \,;\, ABs) | \varnothing \rangle \\ \rightarrow \cdots \rightarrow \quad \langle \mathcal{C}(ABs) | \cup_{k=1}^{d} S_a \rangle \\ \rightarrow \qquad \langle B_i | (\cup_{k=1}^{d} S_a) \rangle \end{aligned}$$

is a valid computation prefix of $\mathcal{C}(tell(a(1))^{||q} \,;\, ABs)$, which can only be continued by failing suffixes. However P induces the following computation prefix P' for $\mathcal{C}(tell(a(1))^{||q} \,;\, (ABs + tell(a(1))))$ which admits only successful computations:

$$\begin{aligned} P' = \langle \mathcal{C}(tell(a(1))^{||d} \,;\, (ABs + tell(a(1)))) | \varnothing \rangle \\ \rightarrow \cdots \rightarrow \quad \langle \mathcal{C}(ABs + tell(a(1))) | \cup_{k=1}^{d} S_a \rangle \\ \rightarrow \qquad \langle B_i | (\cup_{k=1}^{d} S_a) \rangle \end{aligned}$$

leading to the contradiction.

Secondly, the proof that $\{u_1, \cdots, u_q\} \cap S'_b = \varnothing$ is established similarly by considering S'_b instead of S_a and $tell(b(1))$ instead of $tell(a(1))$. □

Let us now establish that $\mathcal{L}_V$(nask,tell) is strictly less expressive than $\mathcal{L}_V$(ask,nask,tell).

Proposition 108. $\mathcal{L}_V$(nask,tell) $< \mathcal{L}_V$(ask,nask,tell).

Proof. By pattern 1 of sublanguage inclusion, one has $\mathcal{L}_V$(nask,tell) $\leqslant \mathcal{L}_V$(ask,nask,tell). Moreover, if we had $\mathcal{L}_V$(ask,nask,tell) $\leqslant \mathcal{L}_V$(nask,tell), then by pattern 3 of non embedding by transitivity we would have $\mathcal{L}_V$(ask,tell) $\leqslant \mathcal{L}_V$(nask,tell), which contradicts proposition 103. □

Let us now establish that $\mathcal{L}_{DB}$(ask,nask,tell) is strictly less expressive than $\mathcal{L}_V$(ask,nask,tell).

Proposition 109. $\mathcal{L}_{DB}$(ask,nask,tell) $< \mathcal{L}_V$(ask,nask,tell).

Proof. **(i)** $\mathcal{L}_{DB}$(ask,nask,tell) $\leqslant \mathcal{L}_V$(ask,nask,tell) results from proposition 97. **(ii)** Let us proceed by contradiction by assuming the existence of a coder $\mathcal{C}$ from $\mathcal{L}_V$(ask,nask,tell) to $\mathcal{L}_{DB}$(ask,nask,tell). Let n be the cumulative occurrences of tokens in *nask* primitives of $\mathcal{C}(tell(a(1)))$.

As $\mathcal{C}(tell(a(1)))$ has only successful computations, let, as in the proof of proposition 100(ii), S_a be the store resulting from one of them. Moreover, as a matter of notation, let the construction $A^{||q}$ denote the parallel composition of q copies of A. As $(tell(b(1)))^{||(n+2)} \,;\, tell(a(1)))$

succeeds as well, let S'_b denote the store resulting from one successful computation of its coding. Consider finally $ABs = ask((a(1), b(n+3)))$ and consider $\mathcal{C}(ABs)$ in its normal form:

$$\begin{array}{l} tell(\overline{t_1})\ ;\ A_1 + \cdots + tell(\overline{t_p})\ ;\ A_p \\ + ask(\overline{u_1})\ ;\ B_1 + \cdots + ask(\overline{u_q})\ ;\ B_q \\ + nask(\overline{v_1})\ ;\ C_1 + \cdots + nask(\overline{v_r})\ ;\ C_r \end{array}$$

We shall establish that: (I) there are no alternatives guarded by $tell(\overline{t_i})$ and $nask(\overline{v_j})$ primitives, and that (II) $\{u_1, \cdots, u_q\} \cap (S_a \cup S'_b) = \varnothing$. Therefore, as computing once more an instance of $\mathcal{C}(tell(b(1))$ in parallel just add copies of tokens already present in $S_a \cup S'_b$, it follows that $\mathcal{C}((tell(b(1))^{||(n+3)}\ ;\ tell(a(1))\ ;\ ABs)$ fails, which is absurd by P_3, since, by construction, $(tell(b(1))^{||(n+3)}\ ;\ tell(a(1))\ ;\ ABs$ has one successful computation.

CASE I: There are no alternatives guarded by a *tell* primitive. The proof proceeds by contradiction as in proposition 105(ii). Assume thus the existence of a $tell(\overline{t_i})\ ;\ A_i$ alternative. Then $D = \langle \mathcal{C}(ABs)|\varnothing\rangle \rightarrow \langle A_i|\{\overline{t_i}\}\rangle$ is a valid computation prefix of $\mathcal{C}(ABs)$ which should deadlock afterwards since $\mathcal{O}(ABs) = \{(\varnothing, \delta^-)\}$. However D is also a valid computation prefix of $\mathcal{C}(ABs + tell(a(1)))$. Hence, $\mathcal{C}(ABs + tell(a(1)))$ admits a failing computation which contradicts the fact that $\mathcal{O}(ABs + tell(a(1))) = \{(\{a\}, \delta^+)\}$.

Moreover, there are no alternatives guarded by a *nask* primitive. In a similar way, assume thus the existence of a $nask(\overline{v_i})\ ;\ C_i$ alternative. Then $D' = \langle \mathcal{C}(ABs)|\varnothing\rangle \rightarrow \langle C_i|\varnothing\rangle$ is a valid computation prefix of $\mathcal{C}(ABs)$ which should deadlock afterwards since $\mathcal{O}(ABs) = \{\varnothing, \delta^-\}$. However D' is also a valid computation prefix of $\mathcal{C}(ABs + tell(a(1)))$. Hence, $\mathcal{C}(ABs + tell(a(1)))$ admits a failing computation which contradicts the fact that $\mathcal{O}(ABs + tell(a(1))) = (\{a\}, \delta^+)$.

CASE II: Let us now prove that $\{u_1, \cdots, u_q\} \cap (S_a \cup S'_b) = \varnothing$. This is proved in two steps by establishing that (1) $\{u_1, \cdots, u_q\} \cap S_a = \varnothing$, and that (2) $\{u_1, \cdots, u_q\} \cap S'_b = \varnothing$.

First let us prove that $\{u_1, \cdots, u_q\} \cap S_a = \varnothing$. Assume $u_i \in S_a$ and let d be the density associated to u_i, namely, $\overline{u_i} = u_i(d)$. Let us observe that each step of the considered computation of $\mathcal{C}(tell(a(1)))$ can be repeated in turn, in as many parallel occurences of it as needed, so that

$$\begin{array}{rl} P = \langle \mathcal{C}(tell(a(1))^{||d}\ ;\ ABs)|\varnothing\rangle & \\ \rightarrow \cdots \rightarrow & \langle \mathcal{C}(ABs)|\ \cup_{k=1}^{d} S_a\rangle \\ \rightarrow & \langle B_i|(\cup_{k=1}^{d} S_a)\rangle \end{array}$$

is a valid computation prefix of $\mathcal{C}(tell(a(1))^{||d}\ ;\ ABs)$, which can only be continued by failing suffixes. However P induces the following computation prefix P' for $\mathcal{C}(tell(a(1))^{||d}\ ;\ (ABs + tell(a(1))))$ which admits only successful computations:

$$\begin{array}{rl} P' = \langle \mathcal{C}(tell(a(1))^{||d}\ ;\ (ABs + tell(a(1))))|\varnothing\rangle & \\ \rightarrow \cdots \rightarrow & \langle \mathcal{C}(ABs + tell(a(1)))|\ \cup_{k=1}^{d} S_a\rangle \\ \rightarrow & \langle B_i|(\cup_{k=1}^{d} S_a)\rangle \end{array}$$

leading to the contradiction.

Secondly, the proof that $\{u_1, \cdots, u_q\} \cap S'_b = \varnothing$ is established similarly by considering S'_b instead of S_a and $tell(b(1))$ instead of $tell(a(1))$. □

$\mathcal{L}_V$(ask,tell) is strictly less expressive than $\mathcal{L}_V$(ask,nask,tell).

Proposition 110. $\mathcal{L}_V(\text{ask,tell}) < \mathcal{L}_V(\text{ask,nask,tell})$

Proof. On the one hand, $\mathcal{L}_V(\text{ask,tell}) \leqslant \mathcal{L}_V(\text{ask,nask,tell})$ results from pattern 1 of sublanguage inclusion. On the other hand, one has $\mathcal{L}_V(\text{ask,nask,tell}) \nleqslant \mathcal{L}_V(\text{ask,tell})$ since otherwise by pattern 3 of non embedding by transitivity $\mathcal{L}_V(\text{nask,tell}) \leqslant \mathcal{L}_V(\text{ask,tell})$, which contradicts proposition 103. □

We now prove that $\mathcal{L}_{DB}$(nask,get,tell) is not comparable to $\mathcal{L}_V$(nask,tell).

Proposition 111. $\mathcal{L}_{DB}(\text{nask,get,tell}) \wr \mathcal{L}_V(\text{nask,tell})$

Proof. **(i)** $\mathcal{L}_{DB}(\text{nask,get,tell}) \nleqslant \mathcal{L}_V(\text{nask,tell})$, otherwise by pattern 3 of non embedding by transitivity, $\mathcal{L}_{DB}(\text{ask,tell}) \leqslant \mathcal{L}_V(\text{nask,tell})$ which contradicts proposition 101.

(ii) $\mathcal{L}_V(\text{nask,tell}) \nleqslant \mathcal{L}_{DB}(\text{nask,get,tell})$ is proved by contradiction, similarly to the proof of $\mathcal{L}_V(\text{nask,tell}) \nleqslant \mathcal{L}_{DB}(\text{ask,nask,tell})$ of proposition 104, which itself extends that of $\mathcal{L}_V(\text{nask,tell}) \nleqslant \mathcal{L}_{DB}(\text{nask,tell})$ of proposition 100.

Given the destructive character of get primitives, we shall enrich them with the saturation technique of the proof of proposition 109, which technically leads to considering the set S'_b instead of the set S_b. Using these notations, we thus fix a token a and reason on two cases, both leading to a contradiction: (I) either there exists a token b such that $S_a \cap S'_b = \varnothing$, (II) or, for any token b, one has $S_a \cap S'_b \neq \varnothing$.

CASE I: there is a token b such that $S_a \cap S'_b = \varnothing$. Consider $AB = nask((a(1), b(1)))$ and $\mathcal{C}(AB)$ in its normal form:

$$\begin{array}{rl} & tell(\overline{t_1}) \,;\, A_1 + \cdots + tell(\overline{t_p}) \,;\, A_p \\ + & get(\overline{u_1}) \,;\, B_1 + \cdots + get(\overline{u_q}) \,;\, B_q \\ + & nask(\overline{v_1}) \,;\, C_1 + \cdots + nask(\overline{v_r}) \,;\, C_r \end{array}$$

As in proposition 109(ii), it is possible to establish that there are no alternatives guarded by a $tell(\overline{t_i})$ primitive : if this was the case then, by posing $A = tell(a(1))$ then AB would point out a deadlock for $A \,;\, (AB + A)$ which only admits successful computations. As in proposition 109(ii) also, it is possible to establish that the v_i's should belong to S_a and to S'_b, which amounts to stating that there are no alternatives guarded by a $nask(\overline{v_j})$ primitive.

Consequently, $\mathcal{C}(AB)$ rewrites as

$$get(\overline{u_1})\ ;\ B_1 + \cdots + get(\overline{u_q})\ ;\ B_q$$

and thus $\mathcal{O}(\mathcal{C}(AB)) = \{(\varnothing, \delta^-)\}$ which, by P_3, contradicts the fact that $\mathcal{O}(AB) = \{(\varnothing, \delta^+)\}$.

CASE II: for any token b, one has $S_a \cap S'_b \neq \varnothing$. By lemma 1, there exists a denumerable set of distinct tokens x_i, also distinct from a, and an integer m, such that $\cap_{i=1}^{m}(S_a \cap S'_{x_i}) \neq \varnothing$ and $[\cap_{i=1}^{m}(S_a \cap S'_{x_i})] \cap (S_a \cap S'_{x_j}) \neq \varnothing$, for $j > m$.

Consider $NT = nask(a(1), x_1(1), \cdots, x_m(1))$ and $\mathcal{C}(NT)$ in the following normal form:

$$\begin{aligned} &tell(\overline{t_1})\ ;\ A_1 + \cdots + tell(\overline{t_p})\ ;\ A_p \\ &+ get(\overline{u_1})\ ;\ B_1 + \cdots + get(\overline{u_q})\ ;\ B_q \\ &+ nask(\overline{v_1})\ ;\ C_1 + \cdots + nask(\overline{v_r})\ ;\ C_r \end{aligned}$$

As for case I, it is possible to prove that there are no alternatives guarded by a $tell(\overline{t_i})$ primitive. It is also possible to establish that

$$\{\overline{v_1}, \cdots, \overline{v_r}\} \subseteq S_a \cap S'_{x_1} \cap \cdots \cap S'_{x_m}$$

Firstly, we have that $\{\overline{v_k}\} \subseteq S_a$, for any k. Otherwise, assume $\{\overline{v_k}\} \nsubseteq S_a$, for some k. Then

$$\begin{aligned} F = \ &\langle \mathcal{C}(tell(a(1))\ ;\ \mathcal{C}(NT) \mid \varnothing\rangle \longrightarrow \cdots \\ &\longrightarrow \langle \mathcal{C}(NT) \mid S_a\rangle \longrightarrow \langle C_k \mid S_a\rangle \end{aligned}$$

would be a valid computation prefix for $\mathcal{C}(tell(a(1))\ ;\ NT)$ which, by property P_3, can only be continued by failing suffixes. However F induces the following computation prefix F' for $\mathcal{C}(tell(a(1))\ ;\ (NT + tell(a(1))))$, and thus a failing computation for it, which by P_3 contradicts the fact that $tell(a(1))\ ;\ (NT + tell(a(1))))$ has only one successful computation.

Secondly, we have that $\{\overline{v_k}\} \subseteq S'_{x_i}$, for any k and i. By contradiction, assume that $\{\overline{v_k}\} \nsubseteq S'_{x_i}$, for some k and i. The proof proceeds similarly by considering $(PP\ ;\ NT)$ instead of $tell(a(1))\ ;\ NT$ and $PP\ ;\ (NT + tell(x_i(1)))$ instead of $tell(a(1))\ ;\ (NT + tell(a(1)))$ with PP being defined as the parallel composition of $n+2$ occurrences of $tell(x_i(1))$ followed by $tell(a(1))$. To that end, note that the computation of $\mathcal{C}(PP)$ leads to the store S'_{x_i} (see propositions 87 and 88)

Consider now $tell(x_{m+1}(1))\ ;\ NT$ and let $\overline{u_i} = u_i(q_i)$ for any i. A possible computation prefix for $\mathcal{C}(tell(x_{m+1}(1))\ ;\ NT)$ is, by P_2, as follows:

$$\langle \mathcal{C}(tell(x_{m+1}(1))\ ;\ \mathcal{C}(NT) \mid \varnothing\rangle \longrightarrow^* \langle \mathcal{C}(NT) \mid S_{x_{m+1}}\rangle$$

Since $tell(x_{m+1}(1))\ ;\ NT$ has a successful computation, and since $\{\overline{v_1}, \cdots, \overline{v_r}\} \subseteq S_a \cap S_{x_1} \cap \cdots S_{x_m} \subseteq S_{x_{m+1}}$ there should exist j such that $\{u_j(q_j)\} \subseteq S_{x_{m+1}}$.

Therefore, as $S_{x_{m+1}} \subseteq S'_{x_{m+1}}$, the following derivation is valid:

$$\begin{aligned} H = \ & \langle \mathcal{C}(tell(x_{m+1}(1))^{||(n+2)}) \ ; \ \mathcal{C}(tell(a(1))) \ ; \ \mathcal{C}(NT) \mid \varnothing \rangle \\ & \longrightarrow^* \langle \mathcal{C}(NT) \mid S'_{x_{m+1}} \rangle \\ & \longrightarrow \langle B_j \mid S'_{x_{m+1}} \backslash \{u_j(q_j)\} \rangle \end{aligned}$$

Moreover, H should be continued by failing suffixes only since $tell(x_{m+1}(1))^{||(n+2)} \ ; \ tell(a(1)) \ ; \ NT$ fails. However, by P_3, this introduces failing computations for $tell(x_{m+1}(1))^{||(n+2)} \ ; \ tell(a(1)) \ ; \ (NT + tell(a(1)))$ whereas this agent has only one successful computation. □

In order to use once more the reasoning of proposition 105, and to make some result available for further proposition 120, we now prove that $\mathcal{L}_V$(ask,tell) is not comparable with $\mathcal{L}_{DB}$(nask,get,tell).

Proposition 112. $\mathcal{L}_V$(ask,tell) $\wr$ $\mathcal{L}_{DB}$(nask,get,tell)

Proof. **(i)** The proof proceeds as in proposition 105, by constructing a successful coded computation for the same failing agent $ask((a(1), b(1)))$ with the alternatives guarded by a nask primitive of the normal form of the coded version treated as the alternatives guarded by a tell primitive. **(ii)** Otherwise by pattern 3 of non embedding by transitivity, $\mathcal{L}_{DB}$(nask,tell) $\leqslant$ $\mathcal{L}_V$(ask,tell) which contradicts proposition 102. □

$\mathcal{L}_{DB}$(nask,get,tell) is not comparable with $\mathcal{L}_V$(ask,nask,tell).

Proposition 113. $\mathcal{L}_{DB}$(nask,get,tell) $\wr$ $\mathcal{L}_V$(ask,nask,tell)

Proof. **(i)** $\mathcal{L}_V$(ask,nask,tell) $\not\leqslant$ $\mathcal{L}_{DB}$(nask,get,tell), otherwise by pattern 3 of non embedding by transitivity, $\mathcal{L}_V$(ask,tell) $\leqslant$ $\mathcal{L}_{DB}$(nask,get,tell) which contradicts proposition 112. **(ii)** $\mathcal{L}_{DB}$(nask,get,tell) $\not\leqslant$ $\mathcal{L}_V$(ask,nask,tell) is proved by contradiction. Consider $tell(t(1)) \ ; \ get(t(1))$, for which $\mathcal{O}((tell(t(1)) \ ; \ get(t(1))) = \{(\varnothing, \delta^+)\}$. Hence, by P_2 and P_3, any computation of $\mathcal{C}(tell(t(1))) \ ; \ \mathcal{C}(get(t(1)))$ is successful. Such a computation is composed of a computation for $\mathcal{C}(tell(t(1)))$ followed by a computation for $\mathcal{C}(get(t(1)))$. As the latter is composed of ask, nask, tell primitives which do not destroy elements on the store, the latter computation can be repeated step by step which yields a successful computation for $\mathcal{C}(tell(t(1))) \ ; \ (\mathcal{C}(get(t(1))) \ || \ \mathcal{C}(get(t(1))))$. However, $\mathcal{O}(tell(t(1)) \ ; \ (get(t(1)) \ || \ get(t(1))) = \{(\varnothing, \delta^-)\}$, which produces the announced contradiction. □

$\mathcal{L}_V$(nask,tell) is not comparable with $\mathcal{L}_{DB}$(get,tell).

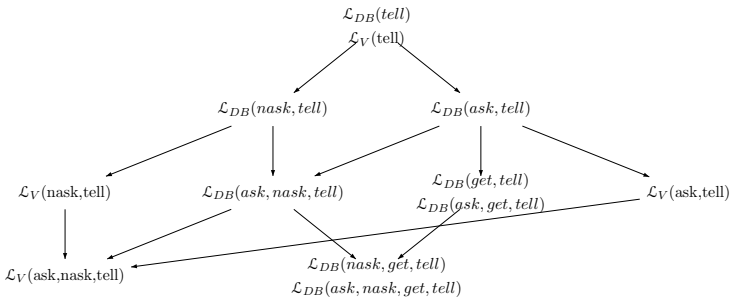


Figure 7.3: Embedding hierarchy of Dense Bach and Vectorized Dense Bach for the tell, ask and nask primitives.

Proposition 114. $\mathcal{L}_V$(nask,tell) $\wr$ $\mathcal{L}_{DB}$(get,tell)

Proof. On the one hand, $\mathcal{L}_V$(nask,tell) $\not\leqslant$ $\mathcal{L}_{DB}$(get,tell). Otherwise, by the pattern 2 of transitivity, $\mathcal{L}_{DB}$(nask,tell) $\leqslant$ $\mathcal{L}_{DB}$(get,tell) which contradicts proposition 68. On the other hand, $\mathcal{L}_{DB}$(get,tell) $\not\leqslant$ $\mathcal{L}_V$(nask,tell). Otherwise $\mathcal{L}_{DB}$(ask,tell) $\leqslant$ $\mathcal{L}_V$(nask,tell) which contradicts proposition 101. □

Figure 7.3 presents a synthesis of the previoulsy obtained expressive relations proved in the two previous subsections. Only the strict relations of expressiveness are shown in the figure. The absence of arrow between two languages means that there is no relation of expressiveness between them.

7.1.4 Retrieving tokens from the store

Let us now include the get primitive in the Vectorized Dense Bach. We first prove that $\mathcal{L}_V$(get,tell) and $\mathcal{L}_V$(ask,get,tell) are equivalent.

Proposition 115. $\mathcal{L}_V$(get,tell) = $\mathcal{L}_V$(ask,get,tell)

Proof. On the one hand, one has $\mathcal{L}_V$(get,tell) $\leqslant$ $\mathcal{L}_V$(ask,get,tell) by pattern 1 of sublanguage inclusion. On the other, by coding $ask(\overrightarrow{t(m)})$ as $get(\overrightarrow{t(m)})$; $tell(\overrightarrow{t(m)})$ and by using the identity as decoder, one has $\mathcal{L}_V$(ask,get,tell) $\leqslant$ $\mathcal{L}_V$(get,tell). □

Let us now establish that $\mathcal{L}_{DB}$(get,tell) is strictly less expressive than $\mathcal{L}_V$(get,tell).

Proposition 116. $\mathcal{L}_{DB}$(get,tell) $< \mathcal{L}_V$(get,tell)

Proof. On the one hand, $\mathcal{L}_{DB}$(get,tell) $\leqslant \mathcal{L}_V$(get,tell) holds by proposition 97. On the other hand, $\mathcal{L}_V$(get,tell) $\nleqslant \mathcal{L}_{DB}$(get,tell) may be proved exactly as in proposition 105(ii), where we replace any occurrence of ask((a(1),b(1))) by get((a(1),b(1))). □

We now prove that $\mathcal{L}_V$(ask,tell) is strictly less expressive than $\mathcal{L}_V$(get,tell).

Proposition 117. $\mathcal{L}_V$(ask,tell) $< \mathcal{L}_V$(get,tell)

Proof. **(i)** Immediate by coding $ask(t(m))$ as $get(t(m))\,;\,tell(t(m))$. **(ii)** By contradiction, consider $A = tell(t(1))\,;\,get(t(1))$. One has $\mathcal{O}(A) = \{(\varnothing, \delta^+)\}$. By P_2 and P_3, any computation of $\mathcal{O}(\mathcal{C}(tell(t(1)))\,;\,\mathcal{C}(get(t(1))))$ is thus successful. Such a computation is composed of a computation for $\mathcal{C}(tell(t(1)))$ followed by a computation for $\mathcal{C}(get(t(1)))$. As $\mathcal{C}(get(t(1)))$ is composed of ask and tell primitives and since ask and tell primitives do not destroy elements, this latter computation can be repeated, which yields successful computations for $\mathcal{O}(\mathcal{C}(tell(t(1)))\,;\,\mathcal{C}(get(t(1)))\,;\,\mathcal{C}(get(t(1))))$. However, $\mathcal{O}(tell(t(1))\,;\,get(t(1))\,;\,get(t(1))) = \{(\varnothing, \delta^-)\}$, which leads to the contradiction. □

The five next propositions establish that $\mathcal{L}_V$(get,tell) is not comparable with respectively $\mathcal{L}_{DB}$(nask,tell), $\mathcal{L}_V$(nask,tell), $\mathcal{L}_{DB}$(nask,get,tell), $\mathcal{L}_V$(ask,nask,tell) and $\mathcal{L}_{DB}$(ask,nask,tell).

Let us first prove that $\mathcal{L}_V$(get,tell) is not comparable with $\mathcal{L}_{DB}$(nask,tell).

Proposition 118. $\mathcal{L}_V$(get,tell) $\wr$ $\mathcal{L}_{DB}$(nask,tell)

Proof. **(i)** $\mathcal{L}_V$(get,tell) $\nleqslant \mathcal{L}_{DB}$(nask,tell), otherwise by pattern 3 of non embedding by transitivity, we have $\mathcal{L}_{DB}$(ask,tell) $\leqslant \mathcal{L}_{DB}$(nask,tell) which has been proved impossible in proposition 44. **(ii)** $\mathcal{L}_{DB}$(nask,tell) $\nleqslant \mathcal{L}_V$(get,tell) is proved by contradiction. Consider $A = tell(t(1))\,;\,nask(t(1))$, for which $\mathcal{O}(A) = \{(\{t(1)\}, \delta^-)\}$. Then, by P_2 and P_3, any computation of $\mathcal{C}(tell(t(1)))\,;\,\mathcal{C}(nask(t(1)))$ must fail whereas we shall establish that $\mathcal{C}(tell(t(1)))\,;\,\mathcal{C}(nask(t(1)))$ has a successful computation. Indeed, let us observe that $\mathcal{O}(tell(t(1))) = \{(\{t(1)\}, \delta^+)\}$ and $\mathcal{O}(nask(t(1))) = \{(\varnothing, \delta^+)\}$. For both cases, by P_3, any computation of $\mathcal{C}(tell(t(1)))$ and $\mathcal{C}(nask(t(1)))$ starting on the empty store is successful. Consequently, since $\mathcal{C}(tell(t(1)))$ and $\mathcal{C}(nask(t(1)))$ are composed of get and tell primitives, so are all of their computations starting from any store. Therefore, any (successful) computation of $\mathcal{C}(tell(t(1)))$ starting on the empty store can be continued by a (successful) computation of $\mathcal{C}(nask(t(1)))$, which leads to the contradiction. □

The second proposition proves that $\mathcal{L}_V$(get,tell) is not comparable with $\mathcal{L}_V$(nask,tell).

Proposition 119. $\mathcal{L}_V$(get,tell) $\wr$ $\mathcal{L}_V$(nask,tell)

Proof. **(i)** $\mathcal{L}_V$(get,tell) $\not\leqslant$ $\mathcal{L}_V$(nask,tell), otherwise by pattern 3 of non embedding by transitivity, we have $\mathcal{L}_{DB}$(ask,tell) $\leqslant$ $\mathcal{L}_V$(nask,tell) which contradicts proposition 101. **(ii)** $\mathcal{L}_V$(nask,tell) $\not\leqslant$ $\mathcal{L}_V$(get,tell), otherwise by the same pattern 3 we have $\mathcal{L}_{DB}$(nask,tell) $\leqslant$ $\mathcal{L}_V$(get,tell) which contradicts proposition 118 above. □

The third proposition establishes that $\mathcal{L}_V$(get,tell) is not comparable with $\mathcal{L}_{DB}$(nask,get,tell).

Proposition 120. $\mathcal{L}_V$(get,tell) $\wr$ $\mathcal{L}_{DB}$(nask,get,tell)

Proof. **(i)** $\mathcal{L}_V$(get,tell) $\not\leqslant$ $\mathcal{L}_{DB}$(nask,get,tell), otherwise by proposition 117 we have $\mathcal{L}_V$(ask,tell) $\leqslant$ $\mathcal{L}_{DB}$(nask,get,tell) which contradicts proposition 112. **(ii)** $\mathcal{L}_{DB}$(nask,get,tell) $\not\leqslant$ $\mathcal{L}_V$(get,tell), otherwise by pattern 3 of non embedding by transitivity, $\mathcal{L}_{DB}$(nask,tell) $\leqslant$ $\mathcal{L}_V$(get,tell) which contradicts proposition 118 above. □

The fourth proposition establishes that $\mathcal{L}_V$(get,tell) is not comparable with $\mathcal{L}_V$(ask,nask,tell).

Proposition 121. $\mathcal{L}_V$(get,tell) $\wr$ $\mathcal{L}_V$(ask,nask,tell)

Proof. **(i)** $\mathcal{L}_V$(get,tell) $\not\leqslant$ $\mathcal{L}_V$(ask,nask,tell) is proved by contradiction. Let us first observe that $\mathcal{O}(tell(t(1)) \,;\, get(t(1))) = \{(\varnothing, \delta^+)\}$. By P_2 and P_3 any computation of $(\mathcal{C}(tell(t(1))) \,;\, \mathcal{C}(get(t(1))))$ starting in the empty store is thus successful. By repeating step by step the computation of $\mathcal{C}(get(t(1)))$, this leads to a successful computation of $(\mathcal{C}(tell(t(1))) \,;\, (\mathcal{C}(get(t(1)) \;||\; \mathcal{C}(get(t(1))))))$ starting in the empty store. However, $\mathcal{O}(tell(t(1)) \,;\, (get(t(1)) \;||\; get(t(1)))) = \{(\varnothing, \delta^-)\}$, which leads to the contradiction. **(ii)** $\mathcal{L}_V$(ask,nask,tell) $\not\leqslant$ $\mathcal{L}_V$(get,tell), otherwise by pattern 3 of non embedding by transitivity, $\mathcal{L}_V$(nask,tell) $\leqslant$ $\mathcal{L}_V$(get,tell) which contradicts proposition 119 above. □

The fifth proposition establishes that $\mathcal{L}_V$(get,tell) is not comparable with $\mathcal{L}_{DB}$(ask,nask,tell).

Proposition 122. $\mathcal{L}_V$(get,tell) $\wr$ $\mathcal{L}_{DB}$(ask,nask,tell)

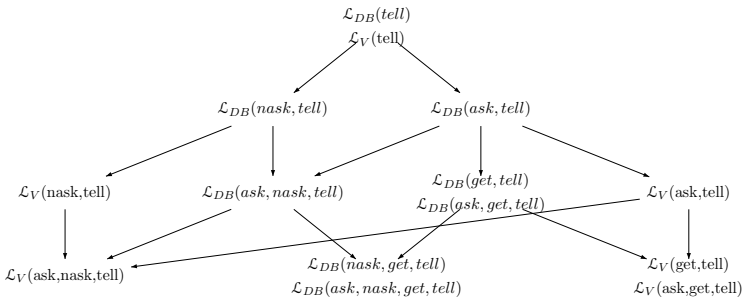


Figure 7.4: Embedding hierarchy of Bach and Vectorized Dense Bach languages for the get primitive in Dense Bach.

Proof. **(i)** $\mathcal{L}_V$(get,tell) $\not\leqslant$ $\mathcal{L}_{DB}$(ask,nask,tell), otherwise by pattern 3 of non embedding by transitivity, $\mathcal{L}_V$(get,tell) $\leqslant$ $\mathcal{L}_V$(ask,nask,tell) which contradicts proposition 121 above. **(ii)** $\mathcal{L}_{DB}$(ask,nask,tell) $\not\leqslant$ $\mathcal{L}_V$(get,tell), otherwise by the same pattern 3 $\mathcal{L}_{DB}$(nask,tell) $\leqslant$ $\mathcal{L}_V$(get,tell) which contradicts proposition 118 above. □

Figure 7.4 adds the expressive relations related to the get primitive in Vectorized Dense Bach to the Figure 7.3 obtained with the tell, ask and nask primitives.

7.1.5 Checking for presence and/or absence when adding and/or retrieving tokens

$\mathcal{L}_V$(nask,get,tell) and $\mathcal{L}_V$(ask,nask,get,tell) are equivalent.

Proposition 123. $\mathcal{L}_V$(nask,get,tell) = $\mathcal{L}_V$(ask,nask,get,tell)

Proof. On the one hand, $\mathcal{L}_V$(nask,get,tell) $\leqslant$ $\mathcal{L}_V$(ask,nask,get,tell) is established by pattern 1 of sublanguage inclusion. On the other hand, to establish $\mathcal{L}_V$(ask,nask,get,tell) $\leqslant$ $\mathcal{L}_V$(nask,get,tell) we shall provide a coder such that the coding of the primitives $ask(\overrightarrow{t(m)})$ and $nask(\overrightarrow{t(m)})$ manipulate different tokens. To that end, as the set of tokens is enumerable, it is possible to associate each of them, say $(\overrightarrow{t(m)})$, to a pair $((\overrightarrow{t_1(m)}), (\overrightarrow{t_2(m)}))$. Given such a coding of tokens, we define the compositional coder $\mathcal{C}$ as follows:

$$\begin{array}{lcl}
\mathcal{C}(ask(\overrightarrow{t(m)})) & = & get(\overrightarrow{t_2(m)})\ ;\ tell(\overrightarrow{t_2(m)}) \\
\mathcal{C}(nask(\overrightarrow{t(m)})) & = & nask(\overrightarrow{t_1(m)}) \\
\mathcal{C}(get(\overrightarrow{t(m)})) & = & get(\overrightarrow{t_2(m)})\ ;\ get(\overrightarrow{t_1(m)}) \\
\mathcal{C}(tell(\overrightarrow{t(m)})) & = & tell(\overrightarrow{t_1(m)})\ ;\ tell(\overrightarrow{t_2(m)})
\end{array}$$

The decoder $\mathcal{D}_c$ is defined as follows: $\mathcal{D}_{el}((\sigma,\delta)) = (\overline{\sigma},\delta)$, where $\overline{\sigma}$ is composed of the tokens $(\overrightarrow{t(m)})$ for which $(\overrightarrow{t_1(m)})$ and $(\overrightarrow{t_2(m)})$ are in σ, the multiplicity of $(\overrightarrow{t(m)})$ being that of pairs $(\overrightarrow{t_1(m)}),(\overrightarrow{t_2(m)})$ in σ. □

$\mathcal{L}_V$(ask,nask,tell) is strictly less expressive than $\mathcal{L}_V$(ask,nask,get,tell), and then from $\mathcal{L}_V$(nask,get,tell), by proposition 123.

Proposition 124. $\mathcal{L}_V$(ask,nask,tell) $< \mathcal{L}_V$(ask,nask,get,tell)

Proof. On the one hand, $\mathcal{L}_V$(ask,nask,tell) $\leqslant \mathcal{L}_V$(ask,nask,get,tell) results from pattern 1 of sublanguage inclusion. On the other hand, $\mathcal{L}_V$(ask,nask,get,tell) $\not\leqslant \mathcal{L}_V$(ask,nask,tell). Otherwise by pattern 3 of non embedding by transitivity, $\mathcal{L}_V$(get,tell) $\leqslant \mathcal{L}_V$(ask,nask,tell), which contradicts proposition 121. □

$\mathcal{L}_V$(get,tell) is strictly less expressive than $\mathcal{L}_V$(nask,get,tell).

Proposition 125. $\mathcal{L}_V$(get,tell) $< \mathcal{L}_V$(nask,get,tell)

Proof. On the one hand, $\mathcal{L}_V$(get,tell) $\leqslant \mathcal{L}_V$(nask,get,tell) results from language inclusion. On the other hand, $\mathcal{L}_V$(nask,get,tell) $\not\leqslant \mathcal{L}_V$(get,tell) is established by contradiction. Consider $tell(t(1))\,;\,nask(t(1))$, for which $\mathcal{O}(tell(t(1))\,;\,nask(t(1))) = \{(\{t(1)\},\delta^-)\}$. Hence, by P_2 and P_3, $\mathcal{C}(tell(t(1)))\,;\,\mathcal{C}(nask(t(1)))$ fails. The contradiction comes then from the fact that at least one computation of $\mathcal{C}(tell(t(1)))\,;\,\mathcal{C}(nask(t(1)))$ starting on the empty store is successful. Indeed, as $\mathcal{O}(tell(t(1))) = \{(\{t(1)\},\delta^+)\}$, any computation of $\mathcal{C}(tell(t(1)))$ starting on the empty store succeeds. Similarly, any computation of $\mathcal{C}(nask(t(1)))$ starting on the empty store succeeds. Moreover, as $\mathcal{C}(nask(t(1)))$ is composed of get and tell primitives only, for any store σ, $\mathcal{C}(nask(t(1)))$ admits at least one successful computation starting on σ. It follows that any computation of $\mathcal{C}(tell(t(1)))$ starting on the empty store can be continued by a (successful) computation of $\mathcal{C}(nask(t(1)))$, which leads to the announced contradiction. □

Finally, $\mathcal{L}_{DB}$(ask,nask,get,tell) can be proved strictly less expressive than $\mathcal{L}_V$(ask,nask,get,tell).

Proposition 126. $\mathcal{L}_{DB}$(ask,nask,get,tell) $< \mathcal{L}_V$(ask,nask,get,tell)

Proof. On the one hand, $\mathcal{L}_{DB}$(ask,nask,get,tell) $\leqslant \mathcal{L}_V$(ask,nask,get,tell) is directly deduced from proposition 97. On the other hand, by pattern 3 of non embedding by transitivity, if one had $\mathcal{L}_V$(ask,nask,get,tell) $\leqslant \mathcal{L}_{DB}$(ask,nask,get,tell) then $\mathcal{L}_V$(get,tell) $\leqslant \mathcal{L}_{DB}$(nask,get,tell) would hold, which contradicts proposition 120. □

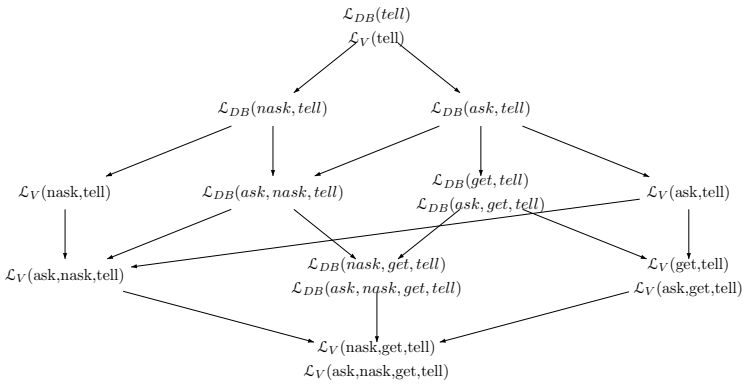


Figure 7.5: Embedding hierarchy of Dense Bach and Vectorized Dense Bach for all the primitives in Dense Bach.

Figure 7.5 presents the full expressive relations related to the ask, nask, get and tell primitives.

7.2 Comparison with MRT

The previous section has studied the expressiveness comparison between the language Dense Bach and the Vectorized Dense Bach language. With this property, this evolution of Dense Bach provides the language with a property close to the capacity of the multi-set language to manipulate atomically many tokens. It is then logical to compare those two languages from the point of view of the expressiveness. This is the subject of this second section.

7.2.1 Generic patterns and results

To develop the different proofs, when possible, we make again use of the different patterns of demonstration emphasized in our previous expressiveness studies, i.e. the pattern 1 of sublanguage inclusion, the pattern 2 of transitivity, and its contraposition in pattern 3, establishing the non-embedding by transitivity.

Moreover, a first observation establishes that Dense Bach with Distributed Density sublanguages are embedded in the corresponding multi-set rewriting sublanguages.

Proposition 127. $\mathcal{L}_V(\chi) \leqslant \mathcal{L}_{MR}(\chi)$, for any subset of χ of primitives.

Proof. Immediate by defining the coder as follows:

$$\begin{aligned}
\mathcal{C}(tell((t_1(m_1),\ldots,t_k(m_k)))) &= (\{\}, \{\underbrace{+t_1,\cdots,+t_1}_{m_1\ times},\cdots,\underbrace{+t_k,\cdots,+t_k}_{m_k\ times}\})\\
\mathcal{C}(ask((t_1(m_1),\ldots,t_k(m_k))) &= (\{\underbrace{+t_1,\cdots,+t_1}_{m_1\ times},\cdots,\underbrace{+t_k,\cdots,+t_k}_{m_k\ times}\}, \{\})\\
\mathcal{C}(get((t_1(m_1),\ldots,t_k(m_k)))) &= (\{\underbrace{+t_1,\cdots,+t_1}_{m_1\ times},\cdots,\underbrace{+t_k,\cdots,+t_k}_{m_k\ times}\}, \{\underbrace{-t_1,\cdots,-t_1}_{m_1\ times},\cdots,\underbrace{-t_k,\cdots,-t_k}_{m_k\ times}\})\\
\mathcal{C}(nask((t_1(m_1),\ldots,t_k(m_k)))) &= (\{\underbrace{-t_1,\cdots,-t_1}_{m_1\ times},\cdots,\underbrace{-t_k,\cdots,-t_k}_{m_k\ times}\}, \{\})
\end{aligned}$$

and using the identity as decoder. □

Finally, as for all the previous expressiveness studies, the different propositions are logically grouped following the same schema. We first consider the sublanguage constituted by the tell primitive only. The store being feeded with tokens, the second step is to provide the language with a possibility to question the store about the presence or the absence of tokens on it. Those two capacities result from the introduction of the ask and nask primitives, in addition to the tell primitive. A third important property is then to allow the language to retrieve tokens from the store, by using the get primitive. Finally the last step studies the most complete language, combining the get and tell primitives with the nask and/or ask primitives.

7.2.2 Adding tokens on the store

When only constituted of the tell primitive, $\mathcal{L}_V$(tell) and $\mathcal{L}_{MR}$(tell) are equivalent.

Proposition 128. $\mathcal{L}_{MR}$(tell) and $\mathcal{L}_V$(tell) are equivalent.

Proof. We have $\mathcal{L}_V$(tell) $\leqslant \mathcal{L}_{MR}$(tell) by proposition 127. Furthermore, $\mathcal{L}_{MR}$(tell) $\leqslant \mathcal{L}_V$(tell) is established by coding any tell primitive of $\mathcal{L}_{MR}$(tell) as the composition of their dense versions : $\mathcal{C}(\{\}, \{\underbrace{+t_1,\cdots,+t_1}_{m_1\ times},\cdots,\underbrace{+t_k,\cdots,+t_k}_{m_k\ times}\}) = tell((t_1(m_1),\cdots,t_k(m_k)))$. □

7.2.3 Checking for presence and/or absence when adding tokens

Let us now consider the introduction of a questioning about the state of the store, regarding the presence or the absence of tokens on it. As a result of the expressiveness hierarchy obtained in section 7.1.1 and synthesized in Figure 7.5, it also comes that both languages $\mathcal{L}_V$(ask,tell) and $\mathcal{L}_V$(nask,tell) are strictly more expressive than $\mathcal{L}_{MR}$(tell) since both have been established

strictly more expressive than $\mathcal{L}_V$(tell).

An interesting result is that $\mathcal{L}_V$(ask,tell) is as expressive as $\mathcal{L}_{MR}$(ask,tell).

Proposition 129. $\mathcal{L}_V$(ask,tell) = $\mathcal{L}_{MR}$(ask,tell)

Proof. **(i)** On the one hand, $\mathcal{L}_V$(ask,tell) $\leqslant$ $\mathcal{L}_{MR}$(ask,tell), by proposition 127. **(ii)** On the other hand, $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_V$(ask,tell) is established by noting that any agent of $\mathcal{L}_{MR}$(ask,tell) can be simulated by an agent of $\mathcal{L}_{MR}$(ask) followed by an agent of $\mathcal{L}_{MR}$(tell). For instance, $\{\underbrace{+a_1,\cdots,+a_1}_{\alpha_1 \text{ times}},\cdots,\underbrace{+a_n,\cdots,+a_n}_{\alpha_n \text{ times}}\},\{\underbrace{+b_1,\cdots,+b_1}_{\beta_1 \text{ times}},\cdots,\underbrace{+b_m,\cdots,+b_m}_{\beta_m \text{ times}}\}$ can be simulated by $(\{\underbrace{+a_1,\cdots,+a_1}_{\alpha_1 \text{ times}},\cdots,\underbrace{+a_n,\cdots,+a_n}_{\alpha_n \text{ times}}\},\{\});(\{\},\{\underbrace{+b_1,\cdots,+b_1}_{\beta_1 \text{ times}},\cdots,\underbrace{+b_m,\cdots,+b_m}_{\beta_m \text{ times}}\})$. As the reader will easily check it, the crucial property is that no agent of $\mathcal{L}_{MR}$(ask,tell) can perturbate any other since its construction only consists in enriching the store, which does not perturbate the atomicity of the test of any agent of $\mathcal{L}_{MR}$(ask,tell). As a result, by inverting the coding of the proof of the proposition 127, it is possible to simulate any agent of $\mathcal{L}_{MR}$(ask,tell) by a sequence of agents of $\mathcal{L}_V$(ask,tell) . For instance, the above agent is coded as $ask((a_1(\alpha_1),\cdots,a_n(\alpha_n)));tell((b_1(\beta_1),\cdots,b_m(\beta_m)))$.

□

Proof technique 5. The reasoning used to establish the above proposition is used once more to prove propositions 134, which demonstrates that $\mathcal{L}_{MR}$(ask,tell) $<$ $\mathcal{L}_V$(ask,nask,tell).

In contrast, $\mathcal{L}_V$(nask,tell) is strictly less expressive than $\mathcal{L}_{MR}$(nask,tell).

Proposition 130. $\mathcal{L}_V$(nask,tell) $<$ $\mathcal{L}_{MR}$(nask,tell).

Proof. **(i)** On the one hand, $\mathcal{L}_V$(nask,tell) $\leqslant$ $\mathcal{L}_{MR}$(nask,tell) holds by proposition 127.

(ii) On the other hand, $\mathcal{L}_{MR}$(nask,tell) $\not\leqslant$ $\mathcal{L}_V$(nask,tell) is proved by considering agent AB = $(\{-a\},\{+b\})$ and agent BA = $(\{-b\},\{+a\})$, with $\mathcal{O}(AB \parallel BA) = \{(\varnothing,\delta^-)\}$. The proof proceeds by contradiction, by assuming the existence of a coder $\mathcal{C}$ with $\mathcal{C}(AB)$ in normal form [BJ98], and thus written as $tell(\overrightarrow{t_1});A_1+\cdots+tell(\overrightarrow{t_p});A_p+nask(\overrightarrow{u_1});B_1+\cdots+nask(\overrightarrow{u_q});B_q$. In this expression we will establish that there is no alternative guarded by a $tell(\overrightarrow{t_i})$ operation and no alternative guarded by a $nask(\overrightarrow{u_j})$ operation either, which is impossible since $\mathcal{C}(AB)$ must contain at least one primitive. We notice that the coding of $\mathcal{C}(AB \parallel BA)$ can be written as $\mathcal{C}(AB) \parallel \mathcal{C}(BA)$ by P_2.

Let us first establish that there is no alternative guarded by a $tell(\overrightarrow{t_i})$ operation. Indeed if there is an alternative guarded, say by $tell(\overrightarrow{t_i})$, then $D = \langle \mathcal{C}(AB \parallel BA)|\varnothing\rangle \rightarrow$

$\langle (A_i \parallel \mathcal{C}(BA))|\{\overrightarrow{t_i}\}\rangle$ is a valid computation prefix of $\mathcal{C}(AB \parallel BA)$. It should deadlock afterwards since $\mathcal{O}(AB \parallel BA) = (\varnothing, \delta^-)$. However D is also a valid computation prefix of $\mathcal{C}((AB \parallel BA) + (\{\}, \{+a\}))$. Hence, $\mathcal{C}((AB \parallel BA) + (\{\}, \{+a\}))$ admits a failing computation which contradicts the fact that $\mathcal{O}((AB \parallel BA) + (\{\}, \{+a\})) = (\{a\}, \delta^+)$.

Secondly we establish that there is no alternative guarded by a $nask(\overrightarrow{u_j})$ operation. Indeed starting from the empty store, if there is an alternative guarded, say by $nask(\overrightarrow{u_j})$, then $D = \langle \mathcal{C}(AB \parallel BA)|\varnothing\rangle \rightarrow \langle (B_j \parallel \mathcal{C}(BA))|\{\overline{t_i}\}\rangle$ is a valid computation prefix of $\mathcal{C}(AB \parallel BA)$. It should deadlock afterwards since $\mathcal{O}(AB \parallel BA) = (\varnothing, \delta^-)$. However D is also a valid computation prefix of $\mathcal{C}((AB \parallel BA) + (\{\}, \{+a\}))$. Hence, $\mathcal{C}((AB \parallel BA) + (\{\}, \{+a\}))$ admits a failing computation which contradicts the fact that $\mathcal{O}((AB \parallel BA) + (\{\}, \{+a\})) = (\{a\}, \delta^+)$.

□

Proof technique 6. The reasoning used to establish the previous proposition is also used to prove propositions 133 and 139, where it is stated that $\mathcal{L}_{MR}$(nask,tell) $\nleqslant \mathcal{L}_V$(ask,nask,tell) and $\mathcal{L}_{MR}$(nask,tell) $\nleqslant \mathcal{L}_V$(nask,get,tell).

$\mathcal{L}_{MR}$(ask,tell) and $\mathcal{L}_V$(nask,tell) are not comparable with each other, and so are $\mathcal{L}_{MR}$(nask,tell) and $\mathcal{L}_V$(ask,tell).

Proposition 131. $\mathcal{L}_{MR}$(ask,tell) $\wr \mathcal{L}_V$(nask,tell)

Proof. On the one hand, we have that $\mathcal{L}_{MR}$(ask,tell) $\nleqslant \mathcal{L}_V$(nask,tell). Otherwise, by pattern 3 of non embedding by transitivity, we have $\mathcal{L}_V$(ask,tell) $\leqslant \mathcal{L}_V$(nask,tell) which has been proved impossible in proposition 103. On the other hand, $\mathcal{L}_V$(nask,tell) $\nleqslant \mathcal{L}_{MR}$(ask,tell) is established by employing the same reasoning as the one used in the second part of the proof of proposition 103, using dense vector notations. □

Proposition 132. $\mathcal{L}_{MR}$(nask,tell) $\wr \mathcal{L}_V$(ask,tell)

Proof. On the one hand, $\mathcal{L}_{MR}$(nask,tell) $\nleqslant \mathcal{L}_V$(ask,tell) holds. Otherwise, by pattern 3 of non embedding by transitivity, we have $\mathcal{L}_V$(nask,tell) $\leqslant \mathcal{L}_V$(ask,tell) which has been proved impossible in proposition 103. On the other hand, $\mathcal{L}_V$(ask,tell) $\nleqslant \mathcal{L}_{MR}$(nask,tell) is established by contradiction by considering $tell(t(1))\ ;\ ask(t(1)))$. Indeed, by noting that $\mathcal{O}(tell(t(1))\ ;\ ask(t(1))) = \{(\{t(1)\}, \delta^+)\}$, the reasoning developped in proposition 101 can be followed, which leads to the contradiction. □

We now prove that $\mathcal{L}_V$(ask,nask,tell) and $\mathcal{L}_{MR}$(nask,tell) are not comparable with each other.

Proposition 133. $\mathcal{L}_V$(ask,nask,tell) $\wr$ $\mathcal{L}_{MR}$(nask,tell)

Proof. **(i)** We have that $\mathcal{L}_V$(ask,nask,tell) $\nleqslant$ $\mathcal{L}_{MR}$(nask,tell). Otherwise, by pattern 3 of non embedding by transitivity, $\mathcal{L}_V$(ask,tell) $\leqslant$ $\mathcal{L}_{MR}$(nask,tell), which has been proved impossible in proposition 132.

(ii) On the other hand, the proof of $\mathcal{L}_{MR}$(nask,tell) $\nleqslant$ $\mathcal{L}_V$(ask,nask,tell) is an extension of the proof used in proposition 130 with normal forms extended with *ask* primitives. It is established by considering agent AB $= (\{-a\},\{+b\})$ and agent BA $= (\{-b\},\{+a\})$, with $\mathcal{O}((\{-a\},\{+b\}) \parallel (\{-b\},\{+a\})) = \{(\varnothing,\delta^-)\}$. The proof proceeds by contradiction, by assuming the existence of a coder $\mathcal{C}$ with $\mathcal{C}(AB)$ in normal form [BJ98], and thus written as $tell(\overrightarrow{t_1}); A_1 + \cdots + tell(\overrightarrow{t_p}); A_p + ask(\overrightarrow{u_1}); B_1 + \cdots + ask(\overrightarrow{u_q}); B_q + nask(\overrightarrow{v_1}); C_1 + \cdots + nask(\overrightarrow{v_r}); C_r$. In this expression we will establish that there is no alternative guarded by a $tell(\overrightarrow{t_i})$ operation or guarded by a $ask(\overrightarrow{u_j})$, or guarded by a $nask(\overrightarrow{v_k})$ operation either, which is impossible since $\mathcal{C}(AB)$ must contain at least one primitive. We notice that the coding of $\mathcal{C}(AB \parallel BA)$ can be written as $\mathcal{C}(AB) \parallel \mathcal{C}(BA)$ by P_2.

Let us first establish that there is no alternative guarded by a $tell(\overrightarrow{t_i})$ operation. Indeed if there is an alternative guarded, say by $tell(\overrightarrow{t_i})$, then $D = \langle \mathcal{C}(AB \parallel BA) | \varnothing \rangle \rightarrow \langle (A_i \parallel \mathcal{C}(BA)) | \{\overrightarrow{t_i}\} \rangle$ is a valid computation prefix of $\mathcal{C}(AB \parallel BA)$. It should deadlock afterwards since $\mathcal{O}(AB \parallel BA) = \{(\varnothing,\delta^-)\}$. However D is also a valid computation prefix of $\mathcal{C}((AB \parallel BA) + (\{\},\{+a\}))$. Hence, $\mathcal{C}((AB \parallel BA) + (\{\},\{+a\}))$ admits a failing computation which contradicts the fact that $\mathcal{O}((AB \parallel BA) + (\{\},\{+a\})) = (\{a\},\delta^+)$.

Secondly we establish that there is no alternative guarded by a $nask(\overrightarrow{v_j})$ operation. Indeed starting from the empty store, if there is an alternative guarded, say by $nask(\overrightarrow{v_j})$, then $D = \langle \mathcal{C}(AB \parallel BA) | \varnothing \rangle \rightarrow \langle (B_j \parallel \mathcal{C}(BA)) | \varnothing \rangle$ is a valid computation prefix of $\mathcal{C}(AB \parallel BA)$. It should deadlock afterwards since $\mathcal{O}(AB \parallel BA) = \{(\varnothing,\delta^-)\}$. However D is also a valid computation prefix of $\mathcal{C}((AB \parallel BA) + (\{\},\{+a\}))$. Hence, $\mathcal{C}((AB \parallel BA) + (\{\},\{+a\}))$ admits a failing computation which contradicts the fact that $\mathcal{O}((AB \parallel BA) + (\{\},\{+a\})) = (\{a\},\delta^+)$.

Then $\mathcal{C}(AB)$ reduces to

$$ask(\overrightarrow{u_1}) \; ; \; B_1 + \cdots + ask(\overrightarrow{u_q}) \; ; \; B_q$$

which thus fails on the empty store whereas $\mathcal{O}(AB) = \{(\varnothing,\delta^+)\}$, providing the contradiction.

□

Let us now prove that $\mathcal{L}_{MR}$(ask,tell) is strictly less expressive than $\mathcal{L}_V$(ask,nask,tell).

Proposition 134. $\mathcal{L}_{MR}$(ask,tell) $<$ $\mathcal{L}_V$(ask,nask,tell)

Proof. **(i)** On the one hand, thanks to proposition 129, $\mathcal{L}_{MR}$(ask,tell) = $\mathcal{L}_V$(ask,tell) $\leqslant$ $\mathcal{L}_V$(ask,nask,tell) and thus $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_V$(ask,nask,tell).

(ii) On the other hand, $\mathcal{L}_V$(ask,nask,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(ask,tell), since otherwise, $\mathcal{L}_V$(nask,tell) $\leqslant$ $\mathcal{L}_V$(ask,nask,tell) $\leqslant$ $\mathcal{L}_{MR}$(ask,tell), which has been proved impossible in proposition 131.

□

We now prove that $\mathcal{L}_V$(ask,nask,tell) is stricly less expressive than $\mathcal{L}_{MR}$(ask,nask,tell).

Proposition 135. $\mathcal{L}_V$(ask,nask,tell) $<$ $\mathcal{L}_{MR}$(ask,nask,tell)

Proof. **(i)** On the one hand, the fact that $\mathcal{L}_V$(ask,nask,tell) $\leqslant$ $\mathcal{L}_{MR}$(ask,nask,tell) is immediate by proposition 127. **(ii)** On the other hand, $\mathcal{L}_{MR}$(ask,nask,tell) $\not\leqslant$ $\mathcal{L}_V$(ask,nask,tell). Otherwise, by pattern 3 of non embedding by transitivity, from $\mathcal{L}_{MR}$(nask,tell) $\leqslant$ $\mathcal{L}_{MR}$(ask,nask,tell), one would get $\mathcal{L}_{MR}$(nask,tell) $\leqslant$ $\mathcal{L}_V$(ask,nask,tell), which has been proved impossible in proposition 133. □

Proposition 136. $\mathcal{L}_V$(get,tell) $\wr$ $\mathcal{L}_{MR}$(ask,tell)

Proof. **(i)** The reasoning used to prove that $\mathcal{L}_V$(get,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(ask,tell) is the same as the one used in the first part of the proof of proposition 105, with dense vector notations. It works by contradiction and by establishing that $tell(t(1))$; $get(t(1))$ can produce a successful computation for $\mathcal{C}(tell(t(1)))$; $\mathcal{C}(get(t(1)))$; $\mathcal{C}(get(t(1))))$, which can obviously not be the case.

(ii) On the other hand, $\mathcal{L}_{MR}$(ask,tell) $\not\leqslant$ $\mathcal{L}_V$(get,tell) is proved by considering agent AB = $(\{+a,+b\},\{\})$, with $\mathcal{O}((\{+a,+b\},\{\})) = \{(\varnothing,\delta^-)\}$. The proof proceeds by contradiction, by assuming the existence of a coder $\mathcal{C}$ with $\mathcal{C}(AB)$ in normal form [BJ98], and thus written as $tell(\overrightarrow{t_1}); A_1 + \cdots + tell(\overrightarrow{t_p}); A_p + get(\overrightarrow{u_1}); B_1 + \cdots + get(\overrightarrow{u_q}); B_q$, where t_i and u_j denote dense vectors $\overrightarrow{t_i}$ and $\overrightarrow{u_j}$ associated with a density.

The proof proceeds by establishing (I) that there is no alternative guarded by a $tell(\overrightarrow{t_i})$ operation, and (II) that there is no alternative guarded by a $get(\overrightarrow{u_j})$ operation, in which case, $\mathcal{C}(AB)$ is equivalent to an empty statement, which is impossible since it is composed of at least one primitive.

STEP I: Let us first establish that there is no alternative guarded by a $tell(\overrightarrow{t_i})$ operation. Otherwise such alternative would point out a failing computation for $\mathcal{C}(AB + (\{\},\{+a\}))$, contradicting the fact that $\mathcal{O}(AB + (\{\},\{+a\})) = \{(\{a\},\delta^+)\}$.

STEP II: Let us now establish that there is no alternative guarded by a $get(\overrightarrow{u_j})$ operation. To that end, let us first consider two auxiliary computations. As $\mathcal{O}((\{\},\{+a\})) = \{(\{a\},\delta^+)\}$, any computation of $\mathcal{C}((\{\},\{+a\}))$ starting in the empty store succeeds. Let $\langle((\{\},\{+a\}))|\varnothing\rangle \rightarrow \cdots \rightarrow \langle E|\{a_1,\ldots,a_m\}\rangle$ be such a computation. Similarly, let $\langle((\{\},\{+b\}))|\varnothing\rangle \rightarrow \cdots \rightarrow$

$\langle E|\{b_1,\ldots,b_n\}\rangle$ be one computation of $\mathcal{C}((\{\},\{+b\}))$. The proof of the claim proceeds in two steps: first by establishing none of the u_i's belong to $\{a_1,\ldots,a_m\}\cup\{b_1,\ldots,b_n\}$ and second by deriving a contradiction therefrom.

First let us prove that none of the u_i's belong to $\{a_1,\ldots,a_m\}$ (the proof for the b_j's being similar). By contradiction, assume that $u_i = a_k$ for some k and that q is the density associated with u_i, namely, $\overrightarrow{u_i} = u_i(q)$. Let us observe that, since it is in $\mathcal{L}_V$(get,tell), the considered computation of $\mathcal{C}((\{\},\{+a\}))$ can be repeated in parallel, as many times as needed. As a result, if, for an agent A and integer n, the notation $A^{||n}$ denotes the parallel composition of n copies of A and if for a token t, the notation t^n in a multiset denotes n occurrences of t, then $D' = \langle\mathcal{C}((\{\},\{+a\})^{||q};AB)|\varnothing\rangle \to\cdots\to\langle\mathcal{C}(AB)|\{a_1^q,\ldots,a_m^q\}\rangle\to\langle B_j|\{a_1^q,\ldots,a_m^q\}\rangle$ is a valid computation prefix of $\mathcal{C}((\{\},\{+a\})^{||q};AB)$, which can only be continued by failing suffixes. However D' induces the following computation prefix D'' for $(\{\},\{+a\})^{||q};(AB+(\{+a\},\{\}))$ which as just seen admits only successful computations: $D'' = \langle\mathcal{C}((\{\},\{+a\})^{||q};(AB+(\{+a\},\{\})))|\varnothing\rangle\to\cdots\to\langle\mathcal{C}(AB+(\{+a\},\{\}))|\{a_1^q,\ldots,a_m^q\}\rangle\to\langle B_j|\{a_1^q,\ldots,a_m^q\}\rangle$.

Second, the fact that the $u_i's$ do not belong to $\{a_1,\ldots,a_m\}\cup\{b_1,\ldots,b_n\}$ induces a contradiction. Indeed, if this is the case then $\langle\mathcal{C}((\{\},\{+a\});(\{\},\{+b\});AB)|\varnothing\rangle\to\cdots\to\langle\mathcal{C}((\{\},\{+b\});AB)|\{a_1,\ldots,a_m\}\rangle\to\cdots\to\langle\mathcal{C}(AB)|\{a_1,\ldots,a_m,b_1,\ldots,b_n\}\rangle\not\to$ is a valid failing computation prefix of $\mathcal{C}((\{\},\{+a\});(\{\},\{+b\});AB)$ whereas $(\{\},\{+a\});(\{\},\{+b\});AB$ has only one successful computation. □

We have that $\mathcal{L}_V$(get,tell) is not comparable with $\mathcal{L}_{MR}$(ask,nask,tell)

Proposition 137. $\mathcal{L}_V$(get,tell) $\wr$ $\mathcal{L}_{MR}$(ask,nask,tell)

Proof. On the one hand, $\mathcal{L}_V$(get,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(ask,nask,tell). The proof proceeds as for establishing that $\mathcal{L}_V$(get,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(ask,tell) (see proposition 136) by considering $tell(t(1))$; $get(t(1))$ and $tell(t(1))$; $(get(t(1))$ || $get(t(1)))$, the parallel composition $\mathcal{C}(get(t(1)))$ || $\mathcal{C}(get(t(1)))$ repeating in turn each step of $\mathcal{C}(get(t(1)))$.

On the other hand, $\mathcal{L}_{MR}$(ask,nask,tell) $\not\leqslant$ $\mathcal{L}_V$(get,tell). Otherwise, by pattern 3, one would have that $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_{MR}$(ask,nask,tell) $\leqslant$ $\mathcal{L}_V$(get,tell) which has been proved impossible in proposition 136. □

We now prove that $\mathcal{L}_{MR}$(ask,tell) is not comparable with $\mathcal{L}_V$(nask,get,tell).

Proposition 138. $\mathcal{L}_V$(nask,get,tell) $\wr$ $\mathcal{L}_{MR}$(ask,tell)

Proof. **(i)** On the one hand, $\mathcal{L}_V$(nask,get,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(ask,tell). Otherwise, by the pattern 3 of non embedding by transitivity, $\mathcal{L}_V$(nask,tell) $\leqslant$ $\mathcal{L}_{MR}$(ask,tell) which has been proved impossible in proposition 131(ii).

(ii) On the other hand, $\mathcal{L}_{MR}$(ask,tell) $\not\leqslant$ $\mathcal{L}_{V}$(nask,get,tell) is proved by considering the agent AB $= (\{+a,+b\},\{\})$, for which $\mathcal{O}((\{+a,+b\},\{\})) = \{(\varnothing,\delta^{-})\}$. We then proceed by contradiction using these two tokens. However, the destructive character of *get* primitives coupled to the test for absence of *nask* slightly complicate our task of producing a contradiction. To that end, we shall "saturate" their effect by taking as many instances of codings in parallel and thereby by extending the sets S_b introduced as in the proof of proposition 83(ii).

Let us thus proceed by contradiction by assuming the existence of a coder $\mathcal{C}$. Take two distinct tokens a and b. Let n be the cumulative sum of the densities associated with the *nask* and *get* primitives of $\mathcal{C}((\{\},\{+a\}))$. As $\mathcal{C}((\{\},\{+a\}))$ has only successful computations, let, as in the proof of proposition 83(ii), S_a be the store resulting from one of them. As $(||_{k=1}^{n+2}(\{\},\{+b\}))\ ;\ (\{\},\{+a\}))$ succeeds as well, let S'_b denote the store resulting from one successful computation of its coding. Consider finally $ABs = (\{+a,+b,\cdots,+b\},\{\})$ requesting one a with $n+3$ copies of b and $\mathcal{C}(ABs)$ in its normal form:

$$\begin{array}{l} tell(\overrightarrow{t_1})\ ;\ A_1 + \cdots + tell(\overrightarrow{t_p})\ ;\ A_p \\ + get(\overrightarrow{u_1})\ ;\ B_1 + \cdots + get(\overrightarrow{u_q})\ ;\ B_q \\ + nask(\overrightarrow{v_1})\ ;\ C_1 + \cdots + nask(\overrightarrow{v_r})\ ;\ C_r \end{array}$$

We shall establish (I) that there are no alternatives guarded by *tell* and *nask* primitives, and (II) that $\{u_1,\cdots,u_q\} \cap (S_a \cup S'_b) = \varnothing$. Assuming these two points proved, a contradiction can be produced as follows. Indeed, in view of the saturation provided by the $n+2$ copies of $\mathcal{C}((\{\},\{+b\}))$, adding one more, only adds tokens present in $S_a \cup S'_b$. As a result, $\mathcal{C}(||_{k=1}^{n+3}(\{\},\{+b\}))\ ;\ (\{\},\{+a\})\ ;\ ABs)$ fails whereas $||_{k=1}^{n+3}(\{\},\{+b\}))\ ;\ (\{\},\{+a\})\ ;\ ABs$ has only one successful computation. Hence the contradiction.

STEP I: Let us first establish that there are no alternative guarded by a $tell(\overrightarrow{t_i})$ primitive. The proof proceeds by contradiction as in proposition 136(ii), by pointing out a failing computation for $\mathcal{C}(AB + (\{\},\{+a\}))$, contradicting the fact that $\mathcal{O}(AB + (\{\},\{+a\})) = \{(\{a\},\delta^{+})\}$.

In a similar way there are no alternative guarded by a *nask* primitive. Indeed assuming the existence of a $nask(\overrightarrow{v_i})\ ;\ C_i$ alternative again points out a failing computation for $\mathcal{C}(AB + (\{\},\{+a\}))$, contradicting the fact that $\mathcal{O}(AB + (\{\},\{+a\})) = \{(\{a\},\delta^{+})\}$.

STEP II: Let us now establish that $\{u_1,\cdots,u_q\} \cap (S_a \cup S'_b) = \varnothing$. This is proved in two steps by establishing (1) that $\{u_1,\cdots,u_q\} \cap S_a = \varnothing$, and (2) that $\{u_1,\cdots,u_q\} \cap S'_b = \varnothing$.

First we have that $\{u_1,\cdots,u_q\} \cap S_a = \varnothing$. By contradiction, assume that $u_i \in S_a$ for some i and let q be the density associated with u_i, namely, $\overrightarrow{u_i} = u_i(q)$. Let us observe that each step of the considered computation of $\mathcal{C}((\{\},\{+a\}))$ can be repeated in turn, in as many parallel

occurences of it as needed, so that

$$\begin{array}{rl} P = \langle \mathcal{C}((||_{k=1}^{q}(\{\},\{+a\}))\ ;\ ABs)|\varnothing\rangle & \\ \rightarrow \cdots \rightarrow & \langle ABs|\ \cup_{k=1}^{q} S_a\rangle \\ \rightarrow & \langle B_i|(\cup_{k=1}^{q} S_a)\backslash\{\overrightarrow{u_i}\}\rangle \end{array}$$

is a valid computation prefix of $\mathcal{C}((||_{k=1}^{q}(\{\},\{+a\}))\ ;\ ABs)$, which can only be continued by failing suffixes. However P induces the following computation prefix P' for $\mathcal{C}((||_{k=1}^{q}(\{\},\{+a\}))\ ;\ (ABs + (\{\},\{+a\})))$ which admits only successful computations:

$$\begin{array}{rl} P' = \langle \mathcal{C}((||_{k=1}^{q}(\{\},\{+a\}))\ ;\ (ABs + (\{\},\{+a\})))|\varnothing\rangle & \\ \rightarrow \cdots \rightarrow & \langle \mathcal{C}(ABs + (\{\},\{+a\}))|\ \cup_{k=1}^{q} S_a\rangle \\ \rightarrow & \langle B_i|(\cup_{k=1}^{q} S_a)\backslash\{\overrightarrow{u_i}\}\rangle \end{array}$$

Hence the contradiction.

The fact that $\{u_1,\cdots,u_q\}\cap S'_b = \varnothing$ is proved similarly, by considering S'_b instead of S_a and $(\{\},\{+b\})$ instead of $(\{\},\{+a\})$. □

In order to prove the next proposition, we need to use lemma 2 of Section 3.2.5.

Proposition 139. $\mathcal{L}_V$(nask,get,tell) $\wr$ $\mathcal{L}_{MR}$(nask,tell)

Proof. **(i)** On the one hand, $\mathcal{L}_V$(nask,get,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(nask,tell). Otherwise, by the pattern 3 on non embedding by transitivity, $\mathcal{L}_V$(ask,tell) $\leqslant$ $\mathcal{L}_{MR}$(nask,tell) which contradicts proposition 132.

(ii) On the other hand, $\mathcal{L}_{MR}$(nask,tell) $\not\leqslant$ $\mathcal{L}_V$(nask,get,tell) is established by contradiction.

Given the destructive character of get primitives, we shall enrich them with the saturation technique of the proof of proposition 138 (ii) which technically leads to considering the set S'_b instead of the set S_b defined in the second part of the proof of proposition 130. Using these notations, we thus fix a token a and reason on two cases, both leading to a contradiction: (I) either there exists a token b such that $S_a \cap S'_b = \varnothing$, (II) or, for any token b, one has $S_a \cap S'_b \neq \varnothing$.

CASE I: there is a token b such that $S_a \cap S'_b = \varnothing$. Consider then $AB = (\{-a,-b\},\{\})$ and $\mathcal{C}(AB)$ in its normal form:

$$\begin{array}{rl} & tell(\overrightarrow{t_1})\ ;\ A_1 + \cdots + tell(\overrightarrow{t_p})\ ;\ A_p \\ + & get(\overrightarrow{u_1})\ ;\ B_1 + \cdots + get(\overrightarrow{u_q})\ ;\ B_q \\ + & nask(\overrightarrow{v_1})\ ;\ C_1 + \cdots + nask(\overrightarrow{v_r})\ ;\ C_r \end{array}$$

As in proposition 130(ii), it is possible to establish that there are no alternatives guarded by a $tell(\overrightarrow{t_i})$ primitive : if this was the case then, by posing $A = (\{\},\{+a\})$, the agent AB would point

out a deadlock for $A\ ;\ (AB + A)$ which only admits successful computations. As in proposition 138(ii) also, it is possible to establish that the v_i's should belong to S_a and to S'_b, which amounts to stating that there are no alternatives guarded by a $nask(\overrightarrow{v_j})$ primitive.

Consequently, $\mathcal{C}(AB)$ rewrites as

$$get(\overrightarrow{u_1})\ ;\ B_1 + \cdots + get(\overrightarrow{u_q})\ ;\ B_q$$

and thus $\mathcal{O}(\mathcal{C}(AB)) = \{(\varnothing, \delta^-)\}$ which, by P_3, contradicts the fact that $\mathcal{O}(AB) = \{(\varnothing, \delta^+)\}$.

CASE II: for any token b, one has $S_a \cap S'_b \neq \varnothing$. By Lemma 2 (where S_a plays the role of S and f is defined by $f(x) = S'_x$), there exists a denumerable set of distinct tokens x_i, also distinct from a, and an integer m, such that $\cap_{i=1}^{m}(S_a \cap S'_{x_i}) \neq \varnothing$ and $[\cap_{i=1}^{m}(S_a \cap S'_{x_i})] \cap (S_a \cap S'_{x_j}) \neq \varnothing$, for $j > m$.

Consider $NT = (\{-a, -x_1, \cdots, -x_m\}, \{\})$ and $\mathcal{C}(NT)$ in the following normal form:

$$\begin{array}{l} tell(\overrightarrow{t_1})\ ;\ A_1 + \cdots + tell(\overrightarrow{t_p})\ ;\ A_p \\ + get(\overrightarrow{u_1})\ ;\ B_1 + \cdots + get(\overrightarrow{u_q})\ ;\ B_q \\ + nask(\overrightarrow{v_1})\ ;\ C_1 + \cdots + nask(\overrightarrow{v_r})\ ;\ C_r \end{array}$$

As for case I, it is possible to prove that there are no alternatives guarded by a $tell(t_i)$ primitive. It is also possible to establish that

$$\{v_1, \cdots, v_r\} \subseteq S_a \cap S'_{x_1} \cap \cdots \cap S'_{x_m}$$

Firstly, we have that $v_k \in S_a$, for any k. Otherwise, assume $v_k \notin S_a$, for some k. Then

$$\begin{array}{rl} F = & \langle \mathcal{C}((\{\}, \{+a\}))\ ;\ \mathcal{C}(NT) \mid \varnothing \rangle \longrightarrow \cdots \\ & \longrightarrow \langle \mathcal{C}(NT) \mid S_a \rangle \longrightarrow \langle C_k \mid S_a \rangle \end{array}$$

would be a valid computation prefix for $\mathcal{C}((\{\}, \{+a\})\ ;\ NT)$ which, by property P_3, can only be continued by failing suffixes. However F induces the following computation prefix F' for $\mathcal{C}((\{\}, \{+a\})\ ;\ (NT + (\{\}, \{+a\})))$, and thus a failing computation for it, which by P_3 contradicts the fact that $(\{\}, \{+a\})\ ;\ (NT + (\{\}, \{+a\})))$ has only one successful computation.

Secondly, we have that $v_k \in S'_{x_i}$, for any k and i. By contradiction, assume that $v_k \notin S'_{x_i}$, for some k and i. The proof proceeds similarly by considering $(PP\ ;\ NT)$ instead of $(\{\}, \{+a\})\ ;\ NT)$ and $PP\ ;\ (NT + (\{\}, \{+x_i\}))$ instead of $(\{\}, \{+a\})\ ;\ (NT + (\{\}, \{+a\}))$ with PP being defined as the parallel composition of $n + 2$ occurrences of $(\{\}, \{+x_i\})$ followed by $(\{\}, \{+a\})$. To that end, note that the computation of $\mathcal{C}(PP)$ leads to the store S'_{x_i} (see the proof of proposition 83(ii)).

Consider now $(\{\}, \{+x_{m+1}\})\ ;\ NT$. A possible computation prefix for $\mathcal{C}((\{\}, \{+x_{m+1}\})\ ;\ NT)$ is, by P_2, as follows:

$$\langle \mathcal{C}((\{\},\{+x_{m+1}\}))\ ;\ \mathcal{C}(NT) \mid \varnothing\rangle \longrightarrow^* \langle \mathcal{C}(NT) \mid S_{x_{m+1}}\rangle$$

Since $(\{\},\{+x_{m+1}\})\ ;\ NT$ has a successful computation, and since $\{v_1,\cdots,v_r\} \subseteq S_a \cap S_{x_1} \cap \cdots S_{x_m} \subseteq S_{x_{m+1}}$ there should exist j such that $u_j \in S_{x_{m+1}}$.

Therefore, as $S_{x_{m+1}} \subseteq S'_{x_{m+1}}$, the following derivation is valid:

$$\begin{aligned} H = \ & \langle \mathcal{C}((\ \|\ _{k=1}^{n+2}(\{\},\{+x_{m+1}\}))\ ;\ \mathcal{C}((\{\},\{+a\}))\ ;\ \mathcal{C}(NT) \mid \varnothing\rangle \\ & \longrightarrow^* \langle \mathcal{C}(NT) \mid S'_{x_{m+1}}\rangle \\ & \longrightarrow \langle B_j \mid S'_{x_{m+1}} \backslash \{\overrightarrow{u_j}\}\rangle \end{aligned}$$

Moreover, H should be continued by failing suffixes only since $(\ \|\ _{k=1}^{n+2}(\{\},\{+x_{m+1}\}))\ ;\ (\{\},\{+a\})\ ;\ NT$ fails. However, by P_3, this introduces failing computations for $(\ \|\ _{k=1}^{n+2}(\{\},\{+x_{m+1}\}))\ ;\ (\{\},\{+a\})\ ;\ (NT + (\{\},\{+a\}))$ whereas this agent has only one successful computation. □

We are now in a position to establish that $\mathcal{L}_V$(get,tell) is not comparable with $\mathcal{L}_{MR}$(nask,tell).

Proposition 140. $\mathcal{L}_V$(get,tell) $\wr$ $\mathcal{L}_{MR}$(nask,tell)

Proof. On the one hand, $\mathcal{L}_V$(get,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(nask,tell). Otherwise, by pattern 3, as $\mathcal{L}_V$(ask,tell) $<$ $\mathcal{L}_V$(get,tell) (see proposition 117(i)), one would have $\mathcal{L}_V$(ask,tell) $\leqslant$ $\mathcal{L}_V$(get,tell) $\leqslant$ $\mathcal{L}_{MR}$(nask,tell) which has been proved impossible in proposition 132. On the other hand, $\mathcal{L}_{MR}$(nask,tell) $\not\leqslant$ $\mathcal{L}_V$(get,tell). Otherwise, by pattern 3, we would have $\mathcal{L}_{MR}$(nask,tell) $\leqslant$ $\mathcal{L}_V$(get,tell) $\leqslant$ $\mathcal{L}_V$(nask,get,tell) which has been proved impossible in proposition 139. □

$\mathcal{L}_V$(nask,get,tell) is not comparable with $\mathcal{L}_{MR}$(ask,nask,tell).

Proposition 141. $\mathcal{L}_{MR}$(ask,nask,tell) $\wr$ $\mathcal{L}_V$(nask,get,tell)

Proof. **(i)** On the one hand $\mathcal{L}_{MR}$(ask,nask,tell) $\not\leqslant$ $\mathcal{L}_V$(nask,get,tell). Otherwise, $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_V$(nask,get,tell) which contradicts proposition 138. **(ii)** On the other hand, $\mathcal{L}_V$(nask,get,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(ask,nask,tell). By contradiction, consider $tell(t(1))\ ;\ get(t(1))$. $\mathcal{O}((tell(t(1))\ ;\ get(t(1))) = \{(\varnothing,\delta^+)\}$. Hence any computation of $\mathcal{C}(tell(t(1)))\ ;\ \mathcal{C}(get(t(1)))$ is successful. Such a computation is composed of a computation for $\mathcal{C}(tell(t(1)))$ followed by a computation for $\mathcal{C}(get(t(1)))$. As $\mathcal{C}(get(t(1)))$ is composed of ask, nask, tell primitives which do not destroy elements on the store, the latter computation can be repeated step by step which yields successful computation for $\mathcal{C}(tell(t(1)))\ ;\ (\mathcal{C}(get(t(1)))\ \|\ \mathcal{C}(get(t(1))))$. However, $\mathcal{O}(tell(t(1))\ ;\ (get(t(1))\ \|\ get(t(1))) = \{(\varnothing,\delta^-)\}$. □

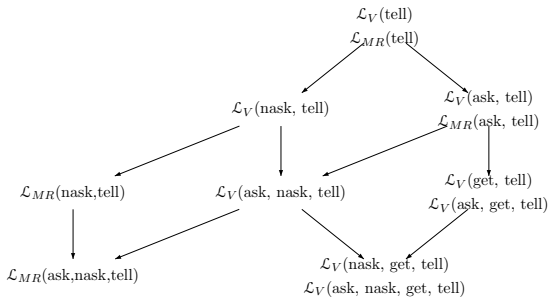


Figure 7.6: Embedding hierarchy of Vectorized Dense Bach and a multi-set rewriting language, considering the presence of the tell, ask and nask primitives in the mutli-set rewriting language.

Figure 7.6 presents the expressive relations established up to now, when only considering the three primitives tell, ask and nask in the multi-set rewriting language.

7.2.4 Retrieving tokens from the store

This section presents the expressive relations when introducing the get primitive in the sublanguages of the multi-set rewriting language. The first proposition establishes that $\mathcal{L}_V$(get,tell) is strictly less expressive than $\mathcal{L}_{MR}$(get,tell).

Proposition 142. $\mathcal{L}_V$(get,tell) $< \mathcal{L}_{MR}$(get,tell)

Proof. **(i)** On the one hand, $\mathcal{L}_V$(get,tell) $\leqslant \mathcal{L}_{MR}$(get,tell) holds by proposition 127. **(ii)** On the other hand, $\mathcal{L}_{MR}$(get,tell) $\nleqslant \mathcal{L}_V$(get,tell) may be proved as for $\mathcal{L}_{MR}$(ask,tell) $\nleqslant \mathcal{L}_V$(get,tell) in proposition 136(ii). Let us thus consider $AB = (\{+a, +b\}, \{-a, -b\})$ and assume that $\mathcal{C}(AB)$ is in normal form (see [BJ98]) and thus is written as $tell(\overrightarrow{t_1}); A_1 + \cdots + tell(\overrightarrow{t_p}); A_p + get(\overrightarrow{u_1}); B_1 + \cdots + get(\overrightarrow{u_q}); B_q$, where $\overline{t_i}$ and $\overline{u_j}$ denote the token t_i and u_j associated with a density.

The proof proceeds by establishing (I) that there is no alternative guarded by a $tell(\overrightarrow{t_i})$ operation, and (II) that there is no alternative guarded by a $get(\overrightarrow{u_j})$ operation, in which case, $\mathcal{C}(AB)$ is equivalent to an empty statement, which is not possible since it should contain at least one primitive.

STEP I: Let us first establish that there is no alternative guarded by a $tell(\overrightarrow{t_i})$ operation. Otherwise it would point out a failing computation for $\mathcal{C}(AB + (\{\}, \{+a\}))$, contradicting the fact that $\mathcal{O}(AB + (\{\}, \{+a\})) = \{(\{a\}, \delta^+)\}$.

STEP II: Let us now establish that there is no alternative guarded by a $get(\overrightarrow{u_j})$ operation. To that end, let us first consider two auxiliary computations: as $\mathcal{O}((\{\}, \{+a\})) = \{(\{a\}, \delta^+)\}$, any

computation of $\mathcal{C}((\{\},\{+a\}))$ starting in the empty store succeeds. Let $\langle((\{\},\{+a\}))|\varnothing\rangle \rightarrow \cdots \rightarrow \langle E|\{a_1,\ldots,a_m\}\rangle$ be such a computation. Similarly, let $\langle((\{\},\{+b\}))|\varnothing\rangle \rightarrow \cdots \rightarrow \langle E|\{b_1,\ldots,b_n\}\rangle$ be one computation of $\mathcal{C}((\{\},\{+b\}))$. The proof of the claim proceeds by establishing, as for proposition 136, that none of the u_i's belong to $\{a_1,\ldots,a_m\} \cup \{b_1,\ldots,b_n\}$, in which case a contradiction occurs from the analysis of $\mathcal{C}((\{\},\{+a\});(\{\},\{+b\});AB)$. As a result, none of the u_i's exist, namely there is no alternative guarded by a $get(\overrightarrow{u_j})$ operation. □

We can now prove that $\mathcal{L}_{MR}$(get,tell) is not comparable respectively with $\mathcal{L}_V$(nask,tell), $\mathcal{L}_V$(nask,get,tell) and $\mathcal{L}_V$(ask,nask,tell).

Proposition 143. $\mathcal{L}_{MR}$(get,tell) $\wr$ $\mathcal{L}_V$(nask,tell)

Proof. On the one hand, $\mathcal{L}_{MR}$(get,tell) $\not\leqslant$ $\mathcal{L}_V$(nask,tell). Otherwise, by pattern 3 on non embedding by transitivity, $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_{MR}$(nask,tell) which has been proved impossible in [BJ03b]. On the other hand, $\mathcal{L}_V$(nask,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(get,tell) is established by contradiction, by considering $tell(t(1))$; $nask(t(1))$. Indeed, one has $\mathcal{O}(tell(t(1)) ; nask(t(1))) = \{(\{t(1)\},\delta^-)\}$ whereas it is possible to establish that $\mathcal{C}(tell(t(1)))$; $\mathcal{C}(nask(t(1)))$ has a successful computation. This is proved by using a reasoning similar to the one used for the second part of proposition 118. □

Proposition 144. $\mathcal{L}_{MR}$(get,tell) $\wr$ $\mathcal{L}_V$(nask,get,tell)

Proof. On the one hand, $\mathcal{L}_{MR}$(get,tell) $\not\leqslant$ $\mathcal{L}_V$(nask,get,tell). Otherwise, by pattern 3 of non embedding by transitivity, as $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_{MR}$(get,tell), we then have $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_V$(nask,get,tell) which has been proved impossible in proposition 138. On the other hand, $\mathcal{L}_V$(nask,get,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(get,tell). Otherwise, by pattern 3, we would have $\mathcal{L}_V$(nask,tell) $\leqslant$ $\mathcal{L}_{MR}$(get,tell) which has been proved impossible in proposition 143. □

Proposition 145. $\mathcal{L}_{MR}$(get,tell) $\wr$ $\mathcal{L}_V$(ask,nask,tell)

Proof. On the one hand, $\mathcal{L}_{MR}$(get,tell) $\not\leqslant$ $\mathcal{L}_V$(ask,nask,tell). Otherwise, by pattern 3, one has $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_{MR}$(get,tell) $\leqslant$ $\mathcal{L}_V$(ask,nask,tell) which has been proved impossible in proposition 134.

On the other hand, $\mathcal{L}_V$(ask,nask,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(get,tell). Otherwise, by pattern 3, one would have $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_{MR}$(ask,nask,tell) $\leqslant$ $\mathcal{L}_V$(get,tell) which has been proved impossible in proposition 136. □

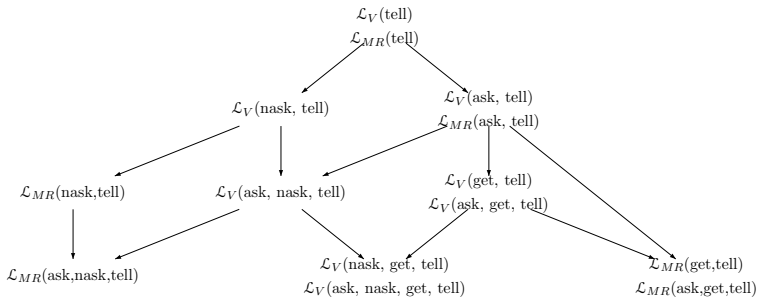


Figure 7.7: Embedding hierarchy of Vecorized Dense Bach and a multi-set rewriting language, considering the presence of the get primitive in the mutli-set rewriting language.

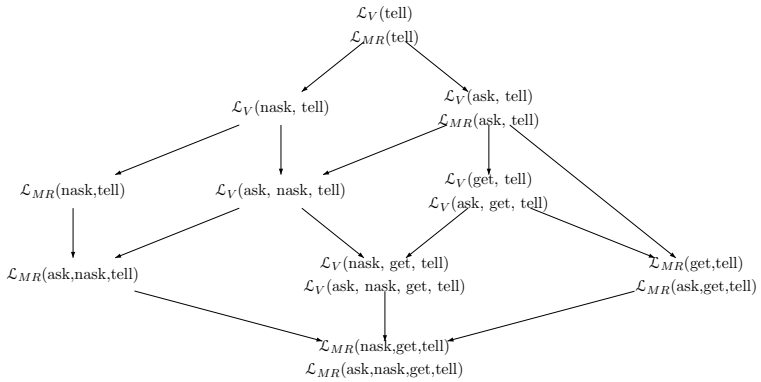


Figure 7.8: Embedding hierarchy of Vectorized Dense Bach and a multi-set rewriting language, considering the presence of all the primitives in the mutli-set rewriting language.

Figure 7.7 complements the previous figure with the introduction of the get primitive inside the subset of the multi-set rewriting language, relating them with the languages of the Dense Bach hierarchy.

7.2.5 Checking for presence and/or absence when adding and/or retrieving tokens

We now prove that $\mathcal{L}_V$(ask,nask,get,tell) is strictly less expressive than $\mathcal{L}_{MR}$(ask,nask,get,tell).

Proposition 146. $\mathcal{L}_V$(ask,nask,get,tell) $< \mathcal{L}_{MR}$(ask,nask,get,tell)

Proof. **(i)** On the one hand, $\mathcal{L}_V$(ask,nask,get,tell) $\leqslant \mathcal{L}_{MR}$(ask,nask,get,tell) is immediate by proposition 127.

(ii) On the other hand, $\mathcal{L}_{MR}$(ask,nask,get,tell) $\nleqslant$ $\mathcal{L}_{V}$(ask,nask,get,tell) is established by contradiction, using pattern 3 of non embedding by transitivity. Indeed, assuming that $\mathcal{L}_{MR}$(ask,nask,get,tell) $\leqslant$ $\mathcal{L}_{V}$(ask,nask,get,tell), as $\mathcal{L}_{V}$(ask,nask,get,tell) = $\mathcal{L}_{V}$(nask,get,tell), one would have $\mathcal{L}_{MR}$(nask, tell) $\leqslant$ $\mathcal{L}_{MR}$(ask,nask,get,tell) $\leqslant$ $\mathcal{L}_{V}$(ask,nask,get,tell) $\leqslant$ $\mathcal{L}_{V}$(nask,get,tell) which has been proved impossible in proposition 139. □

Figure 7.8 presents the most complete view of all the expressiveness relations between the different sublanguages of Vectorized Dense Bach and of the multi-set rewriting language.

7.3 Conclusion

This chapter has studied Vectorized Dense Bach from an expressiveness point of view in terms of the relation of its sublanguages but also with respect to the Dense Bach and MRT languages. The same logical approach as for Dense Bach has been followed. First the proposals have been grouped by considering the feeding of the store with the *tell* primitive. Then the *ask* and *nask* primitives have been introduced in the picture to allow to question the same store about the presence or absence of tokens on it. The *get* primitive has subsequently been introduced providing the possibility of the retrieval of tokens. Finally the language grouping all the primitives has been considered.

The Vectorized Dense Bach language extends the Dense Bach language by permitting to manipulate atomicaly not only dense tokens, but also a list of dense tokens. With such a new behaviour, an increase in expressiveness with respect to Dense Bach has been established. Figure 7.9 illustrates it. The equality for the *tell* primitive is maintained, and we observe again a preservation of the very nature of the *tell*, *ask*, *get* and *nask* primitives. This implies that the hierarchies of the expressiveness relations between the different sublanguages stays similar for Dense Bach and Vector Dense Bach.

Nevertheless concerning the expressiveness relations between Dense Bach and MRT, we observe not only an equality regarding the *tell* primitive, but also between the sublanguages constitued by the *ask* and *tell* primitives.

As for the expressiveness study between BachT, Dense Bach and MRT, a tabulated result is presented in Table 7.1. For every family of language, the different sublanguages are written in line as well as in column. Every intersection mentions the state of the relation, as well as a reference to the proof. In particular, a number represents the number of the proof developed in the thesis.

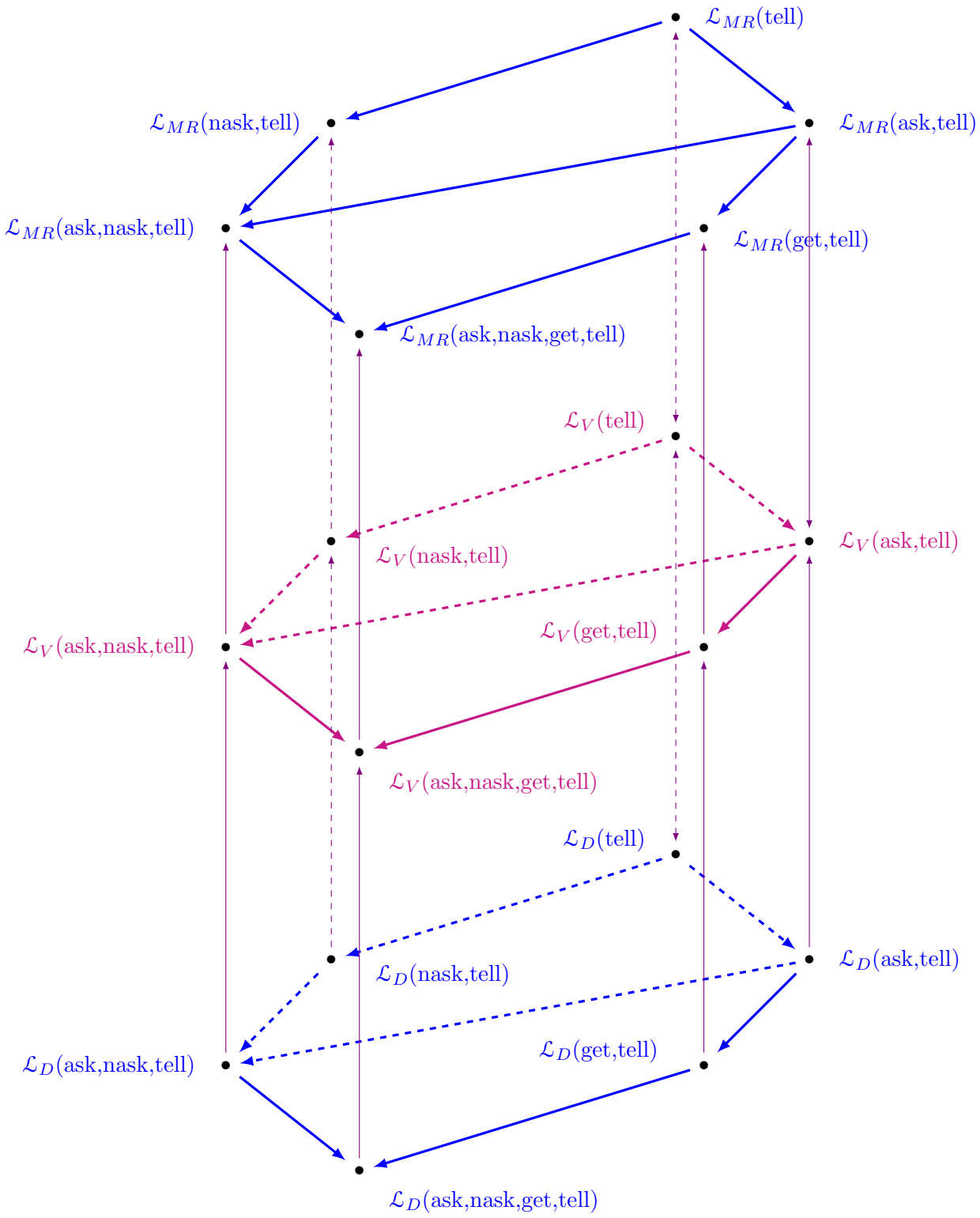


Figure 7.9: Three-dimensional representation of the expressiveness relations between the different sublanguages of Dense Bach, Vectorized Dense Bach and MRT.

Put reduced figure here	$\mathcal{L}_{DB}(tell)$, $\mathcal{L}_V$(tell) $\mathcal{L}_{MR}$(tell)	$\mathcal{L}_{DB}(ask, tell)$	$\mathcal{L}_{DB}(nask, tell)$	$\mathcal{L}_{DB}(get, tell)$ $\mathcal{L}_{DB}(ask, get, tell)$	$\mathcal{L}_{DB}(ask, nask, tell)$	$\mathcal{L}_{DB}(nask, get, tell)$ $\mathcal{L}_{DB}(ask, nask, get, tell)$	$\mathcal{L}_V$(ask,tell)	$\mathcal{L}_V$(nask,tell)	$\mathcal{L}_V$(get,tell) $\mathcal{L}_V$(ask,get,tell)	$\mathcal{L}_V$(ask,nask,tell)	$\mathcal{L}_V$(nask,get,tell) $\mathcal{L}_V$(ask,nask,get,tell)	$\mathcal{L}_{MR}$(ask,tell)	$\mathcal{L}_{MR}$(nask,tell)	$\mathcal{L}_{MR}$(get,tell) $\mathcal{L}_{MR}$(ask,get,tell)	$\mathcal{L}_{MR}$(ask,nask,tell)	$\mathcal{L}_{MR}$(nask,get,tell) $\mathcal{L}_{MR}$(ask,nask,get,tell)
$\mathcal{L}_{DB}$(tell), $\mathcal{L}_V$(tell), $\mathcal{L}_{MR}$(tell)	= 128, 98	≺ 2	≺ 2	≺ 2	≺ 2	≺ 2	≺ 2	≺ 2	≺ 2	≺ 2	≺ 2	≺ 2	≺ 2	≺ 2	≺ 2	≺ 2
$\mathcal{L}_{DB}$(ask,tell)	=	≀ 44	≺ 66	≺ 56	≺ 2	≺ 99	≀ 101	≺ 2	≺ 2	≺ 2	≺ 78	≀ 81	≺ 2	≺ 2	≺ 2	≺ 2
$\mathcal{L}_{DB}$(nask,tell)		=	≀ 68	≺ 54	≺ 2	≀ 102	≺ 100	≀ 118	≺ 2	≺ 2	≀ 80	≺ 79	≀ 92	≺ 2	≺ 2
$\mathcal{L}_{DB}$(get,tell), $\mathcal{L}_{DB}$(ask,get,tell)			= 64	≀ 70	≺ 74	≀ 105	≀ 114	≺ 116	≀ 106	≺ 2	≀ 85	≀ 89	≺ 91	≀ 86	≺ 2
$\mathcal{L}_{DB}$(ask,nask,tell)				=	≺ 73	≀ 107	≀ 104	≀ 122	≺ 109	≺ 2	≀ 83	≀ 82	≀ 94	≺ 84	≺ 2
$\mathcal{L}_{DB}$(nask,get,tell), $\mathcal{L}_{DB}$(ask,nask,get,tell)					= 72	≀ 112	≀ 111	≀ 120	≀ 113	≺ 126	≀ 87	≀ 88	≀ 93	≀ 90	≺ 95
$\mathcal{L}_V$(ask,tell)						=	≀ 103	≺ 117	≺ 110	≺ 2	= 129	≀ 132	≺ 2	≺ 2	≺ 2
$\mathcal{L}_V$(nask,tell)							=	≀ 119	≺ 108	≺ 2	≀ 131	≺ 130	≀ 143	≺ 2	≺ 2
$\mathcal{L}_V$(get,tell), $\mathcal{L}_V$(ask,get,tell)								= 115	≀ 121	≺ 125	≀ 136	≀ 140	≺ 142	≀ 137	≺ 2
$\mathcal{L}_V$(ask,nask,tell)									=	≺ 124	≀ 134	≀ 133	≀ 145	≺ 135	≺ 2
$\mathcal{L}_V$(nask,get,tell), $\mathcal{L}_V$(ask,nask,get,tell)										= 123	≀ 138	≀ 139	≀ 144	≀ 141	≺ 146
$\mathcal{L}_{MR}$(ask,tell)											=	≀ [BJ03b]	≺ [BJ03b]	≺ [BJ03b]	≺ [BJ03b]
$\mathcal{L}_{MR}$(nask,tell)												=	≀ [BJ03b]	≺ [BJ03b]	≺ [BJ03b]
$\mathcal{L}_{MR}$(get,tell), $\mathcal{L}_{MR}$(ask,get,tell)													=	≀ [BJ03b]	≺ [BJ03b]
$\mathcal{L}_{MR}$(ask,nask,tell)														=	≺ [BJ03b]
$\mathcal{L}_{MR}$(nask,get,tell), $\mathcal{L}_{MR}$(ask,nask,get,tell)															=

Table 7.1: Table summarizing the expressiveness comparisons between the different sublanguages of Dense Bach, Vectorized Dense Bach and MRT.

Part III

Programming Aspects

Chapter 8

On the Implementation of Dense Bach

In order to allow the reader to experiment with the dense languages we have introduced in chapters 4 and 5 but also in the aim of arguing for their implementability, we have developed, for each language, an interpreter and a command line simulator, both based on the Scala language.

We turn in this chapter to Dense Bach. As it shares many of the features of BachT and since this language is simpler, we first present in section 8.1 a command line interperter for BachT. We then show in section 8.2 how it can be adapted to produce a command line simulator for BachT and in section 8.3 how it can be extended to generate an interpreter for Dense Bach. Finally, based on these results, a command line simulator for Dense Bach is presented in section 8.4.

8.1 A command-line interpreter for BachT

8.1.1 Introduction

As a first step, in order to allow the reader to experiment with BachT (and later with the other dense languages), our goal is to provide him with a simple interface allowing him to ask for the run of an agent expressed in BachT. As an example, we aim at something as follows :

```
> run "(tell(t);get(u)) || (get(t);tell(u))"

{ t }
{ }
{ u }
{ }
Success
```

Note that such a trace corresponds to the only possible execution. Indeed the computation of the above parallel agent necessarily consists of executing in sequence the $tell(t)$ primitive, yielding the store $\{t\}$, then the $get(t)$ primitive, yielding the empty store, then the $tell(u)$ primitive, yielding the store $\{u\}$ and finally the $get(u)$ primitive, yielding the empty store. In doing so, all the four primitives have been successfully computed, which allows to conclude to a successful computation.

The choice of programming languages to construct such an interface is huge. Two criteria came however rapidly in our mind to select one : first to be able to use the power of functional programming and second to benefit from the graphics facilities offered by object programming.

As regards the first criteria, it is worth observing that the definition of our languages (see for instance definitions 2, 3, 5, 11 and 12) naturally lead to recursive reasonings, which are so naturally handled by functional programming. For the second criteria, object programming has been widely recognized as particularly well suited to build interfaces and moreover is largely promoted by Java, one of the dominant languages.

Given these two criteria, Scala has appeared to us as a good choice since it combines object-oriented programming and functional programming. Additionnally, the attention paid by its developpers to avoid programmers to write what can be implicitly deduced by the type system offers a very concise way of implementing languages.

More precisely, as noted in [MO06] and [MZ06], from the object-oriented flavor, Scala stays close to conventional languages such as Java and C#, sharing with them most of the basic operators, data types, and control structures. Thanks to this, Scala can seamlessly inter-operate with code written in those two languages. Similarly to Smalltalk, Scala considers every value as an object, and every operation as a message send, resulting from the invocation of a method. Scala classes and objects can inherit from Java classes and implement Java interfaces. This facilitates the use of Scala code inside Java framework. From the functional point of view, Scala considers that every function is a value.

Scala supports both styles of abstraction for types and for values: parameterization and abstract members. It has a mechanism of mixin-class composition, which is a form of multiple

```
class Expr
case class bacht_ast_empty_agent() extends Expr
case class bacht_ast_primitive(primitive: String, token: String) extends Expr
case class bacht_ast_agent(op: String, agenti: Expr, agentii: Expr) extends Expr
```

Figure 8.1: The abstract BachT data.scala file

inheritance. Finally Scala allows for decomposition of objects by pattern matching.

After this brief explanation of Scala, we are now ready to present our interpreter. It is composed of three main components : a parser of agents, the implementation of the store and finally a simulator which performs the execution of a BachT agent from the execution of basic primitives. For that latter purpose, it is helpful to represent the structure of a parsed agent in internal structures. This is the purpose of the following abstract data, also depicted in Figure 8.1. Technically, an abstract class, called `Expr`, is first introduced. It is refined in three ways :

- as a case class `bacht_ast_empty_agent` to represent the empty agent E,
- as a case class `bacht_ast_primitive` to represent a primitive in the form of a pair composed of primitive type (tell, ask, nask, get) and of a token
- as a case class `bacht_ast_agent` to represent a composed agent formed from an operator applied to two sub-agents `agenti` and `agentii`.

8.1.2 The parser

As exposed in chapter 33 of [MO10], Scala offers facilities to parse languages. The main ingredients to do so are, on the one hand, a library to define parsers, which subsequently basically allows to define the class `BachTParsers` as inherited from the class `RegexParsers`, parsing regular expressions, and the possibility of applying functions to the result of strings having been parsed. This is technically achieved in three ways :

- firstly, by considering parsers as functions that consume a reader and yield a parse result and by sequencing these consumptions through the ~ operator. For instance, in

  ```
  "tell(" ~ token ~ ")"
  ```

Scala tries to read the string `tell(` then what is defined by the function `token` and finally the string `)`. The value returned by this evaluation is formally an instance of the `~` class, which here can be viewed as a pair, or, in our case, as two embedded pairs, namely a triple.

It is worth noting that repetition can be specified by the `rep` operator. In that case, the value returned is a list.

- secondly, by allowing, through the `^^` construction, to apply a function to the result of a parser, as in

```
("[a-z][0-9a-zA-Z_]*").r ^^ {_.toString}
```

There the regular expression, obtained by `[a-z][0-9a-zA-Z_]*`, is passed to the `toString` function, which transforms it to a string.

- thirdly, by examining a value through a case statement, which allows to perform a matching, as illustrated as follows :

```
"tell("~token~")" ^^ {
        case _ ~ vtoken ~ _  => bacht_ast_primitive("tell",vtoken) }
```

There a string composed of the string "tell(" followed by the result of the function `token` – which turns out to return the string as just explained above with the regular expression – followed by the string ")" is given as the corresponding threefold sequence of strings to be matched to the expression `_ ~ vtoken ~ _`. It is worth noting that `vtoken` is actually a variable which is matched with the corresponding token in case the matching is successful. The underscores denotes different anonymous variables which are respectively used to match the strings "tell(" and ")". In case the matching is successful, the value after the arrow `=>` is given as a result of the parsing. In the above example, a new case class is returned for the primitive tell with `vtoken` as token. It is worth noting that for expressivity purpose, Scala permits to avoid to explicitly write the new statement (which would have been written in Java for instance).

Writing the parser forces us to specify the priorities of the operators – which we have not done when presenting the language in definition 5. Our choice is quite classical : we stipulate that the sequential composition binds more than the parallel composition which itself binds more than the non-deterministic choice operator. As a result and given the fact that left-recursion is to be avoided by Scala parsers, we define agents as follows :

- an agent is a choice-like agent
- a choice-like agent is a parallel-like agent possibly followed by the choice operator followed by a choice-like agent
- a parallel-like agent is a sequential-like agent possibly followed by the parallel operator followed by a parallel-like agent
- a sequential-like agent is a simple agent possibly followed by the sequential operator followed by a sequential-like agent
- a simple agent is either a primitive or an agent enclosed between parentheses.

The code for the parsing of the agents follows directly from this intuition. It is embodied in the functions `compositionChoice`, `compositionPara`, `compositionSeq`, `simpleAgent` and `parenthesizedAgent`. For them, it is worth noting that parsing is applied recursively which forces variables to be instantiated to the result of the parsing of subexpressions. Take the following function as an example :

```
def compositionPara : Parser[Expr] =
                compositionSeq~rep(opPara~compositionPara) ^^ {
      case ag ~ List() => ag
      case agi ~ List(op~agii)  => bacht_ast_agent(op,agi,agii) }
```

The first case of the matching `ag ~ List()` instantiates `ag` to the result of a sequential-like agent which is followed by an empty list, namely which is followed by an empty repetition of the parallel operator followed by a parallel-like agent. In the second case of the matching, `agi` represents the parsing of the sequential-like agent (in a similar way `ag` does for the first case) and `agii` represents the parsing of the repetition of the parallel operator followed by a parallel-like agent. As an example, assume the agent $(tell(t); get(u))||(get(t); tell(u))$ is parsed by the function `compositionPara`. Then the values for `agi` and `agii` are respectively

```
agi = bacht_ast_agent( ";", bach_ast_primitive("tell", "t"),
                           bach_ast_primitive("get", "u"))
agii = bacht_ast_agent( ";", bach_ast_primitive("get", "t"),
                           bach_ast_primitive("tell", "u"))
```

Consequently, it is sufficient to build the structure `bach_ast_agent(op,agi,agii)` to get the expected internal form :

```
bach_ast_agent("||",
                bacht_ast_agent( ";", bach_ast_primitive("tell", "t"),
                                     bach_ast_primitive("get", "u")),
                agii = bacht_ast_agent( ";", bach_ast_primitive("get", "t"),
                                     bach_ast_primitive("tell", "u")))
```

The code of the parser is presented in Figure 8.2. Besides the functions described for the agents, it consists of a function `token` for parsing tokens and of the definition of three values to represent the three operators (non-deterministic choice operator, parallel operator and sequential operator). As the reader will easily notice, we have defined a token as a string composed of at least a small letter ranging between *a* and *z*, possibly followed by a composition of figures between 0 and 9 and/or small or capital letters.

It is practical to define an object instantiating the `BachTParsers` so as to use it directly in the command-line interpreter. This is achieved in the code of Figure 8.3. Two methods are furthermore provided to parse primitives and compositionnally composed agents.

8.1.3 The store

The store is implemented as a mutable map in Scala. Initially empty, it is enriched for each told token by an association of this token with a number representing the number of its occurrences on the store. More precisely, the execution of a *tell* primitive, say *tell(t)* consists in checking whether *t* is already in the map. If it is then the number of occurrences associated with it is simply incremented by one. Otherwise a new association *(t,1)* is added to the map. Dually, the execution of *get(t)* consists in checking first whether *t* is in the map and then whether it is associated with a strictly positive integer. In this case, the execution of *get(t)* consists in decrementing by one the number of associated occurrences. In case one of these two conditions is not met then the *get* primitive cannot be executed. Note that with this simple strategy, a token may appear in the map but with 0 as a number of occurrences associated with it. Hence the implementation has not only to test whether the token appears in the map but also to test whether the associated number of occurrences is more than one. The *ask* primitive has a similar behaviour without removing an instance. Finally the *nask* primitive has an opposite behaviour, succeeding in case the *ask* primitive fails and failing in case it succeeds.

The code for the primitives is presented in Figure 8.4. Two auxiliary functions are presented in Figure 8.5. The first one, *print_store* takes care of the printing of the contents of the store. The second one, *clear_store* aims at resetting the store to the empty map.

```
class BachTParsers extends RegexParsers {

  def token      : Parser[String] = ("[a-z][0-9a-zA-Z_]*").r ^^ {_.toString}

  val opChoice   : Parser[String] = "+"
  val opPara     : Parser[String] = "||"
  val opSeq      : Parser[String] = ";"

  def primitive : Parser[Expr]   = "tell("~token~")" ^^ {
        case _ ~ vtoken ~ _  => bacht_ast_primitive("tell",vtoken) }  |
                                "ask("~token~")" ^^ {
        case _ ~ vtoken ~ _  => bacht_ast_primitive("ask",vtoken) }   |
                                "get("~token~")" ^^ {
        case _ ~ vtoken ~ _  => bacht_ast_primitive("get",vtoken) }   |
                                "nask("~token~")" ^^ {
        case _ ~ vtoken ~ _  => bacht_ast_primitive("nask",vtoken) }

  def agent = compositionChoice

  def compositionChoice : Parser[Expr] = compositionPara~rep(opChoice~compositionChoice)
      ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => bacht_ast_agent(op,agi,agii) }

  def compositionPara : Parser[Expr] = compositionSeq~rep(opPara~compositionPara) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => bacht_ast_agent(op,agi,agii) }

  def compositionSeq : Parser[Expr] = simpleAgent~rep(opSeq~compositionSeq) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => bacht_ast_agent(op,agi,agii) }

  def simpleAgent : Parser[Expr] = primitive | parenthesizedAgent

  def parenthesizedAgent : Parser[Expr] = "("~>agent<~")"

}
```

Figure 8.2: Parser: the class `BachTParsers`

```
object BachTSimulParser extends BachTParsers {

  def parse_primitive(prim: String) = parseAll(primitive,prim) match {
        case Success(result, _) => result
        case failure : NoSuccess => scala.sys.error(failure.msg)
  }

  def parse_agent(ag: String) = parseAll(agent,ag) match {
        case Success(result, _) => result
        case failure : NoSuccess => scala.sys.error(failure.msg)
  }

}
```

Figure 8.3: Parser : the object `BachTSimulParser`

Finally, Figure 8.6 defines the object *bb* of type *BachStore* with a function *reset* as a synonym for the *clear_store* function. Both constructs will be handy for using the command line simulator.

8.1.4 The simulator

The simulator consists in repeatedly executing a transition step, as defined by the operational semantics of section 3.1.2. In our implementation, this boils down to the definition of function `run_one`, which assumes an agent in a parsed form given and which returns a pair composed of a boolean and an agent in parsed form. The boolean aims at specifying whether a transition step has taken place. In this case, the associated agent consists of the agent obtained by the transition step. Otherwise, failure is reported with the given agent as associated agent.

The function assumes a store. It is given as a parameter of the `BachTSimul` in which `run_one` is defined.

The function is defined inductively on the structure of its argument, say `agent`. If it is a primitive, then the `run_one` function simply consists in executing the primitive on the store. This is technically achieved by the `exec_primitive` function, which actually calls the associated primitive function on the store.

If `agent` is a sequentially composed agent ag_i ; ag_{ii}, then the transition step proceeds by trying to execute the first subagent ag_i. Assume this succeeds and delivers ag' as resulting agent. Then the agent returned is ag' ; ag_{ii} in case ag' is not empty or more simply ag_{ii} in case ag' is empty. Of course, the whole computation fails in case ag_i cannot perform a transition step, namely in case `run_one` applied to ag_i fails.

The code for these two first cases is presented in Figure 8.7.

```
import scala.collection.mutable.Map

class BachTStore {

   var theStore = Map[String,Int]()

   def tell(token:String):Boolean = {
      if (theStore.contains(token))
        { theStore(token) = theStore(token) + 1 }
      else
        { theStore = theStore ++ Map(token -> 1) }
      true
   }


   def ask(token:String):Boolean = {
      if (theStore.contains(token))
             if (theStore(token) >= 1) { true }
             else { false }
      else false
   }


   def get(token:String):Boolean = {
      if (theStore.contains(token))
             if (theStore(token) >= 1)
               { theStore(token) = theStore(token) - 1
                 true
               }
             else { false }
      else false
   }


   def nask(token:String):Boolean = {
      if (theStore.contains(token))
             if (theStore(token) >= 1) { false }
             else { true }
      else true
   }


   ...
```

Figure 8.4: The `BachTStore class`

```
class BachTStore {

  ...

   def print_store {
      println("{")
      for ((t,d) <- theStore)
         println ( t + "(" + theStore(t) + ")" )
      println("}")
   }

   def clear_store {
      theStore = Map[String,Int]()
   }

}
```

Figure 8.5: The `BachTStore class` continued

```
object bb extends BachTStore {

   def reset { clear_store }

}
```

Figure 8.6: The `bb` object

The cases of the composed agent by a parallel or choice operator are more subtle. Indeed for both cases one should not always favour the first or second subagent. To avoid that behaviour, we randomly assign 0 or 1 to the `branch_choice` variable and depending upon this value we start by evaluating the first or second subagent. In case of failure, we then evaluate the other one and if both fails we report a failure. In case of success for the parallel composition we determine the resulting agent in a similar way to what we did for the sequentially composed agent. For a composition by the choice operator the tried alternative is simply selected. The code for these two cases is reported in Figures 8.8 and 8.9.

With the one step transition function coded, the simulator mainly consists of a loop which is executed while the current agent is non empty and while failure does not occur. This is materialized in the `bacht_exec_all` function detailed in Figure 8.10. As for the previous component, an object is created to ease the deployment of the command-line interpreter. It defines an `apply` function which essentially consists of executing the function `bacht_exec_all` on the parsed agent. Two other functions `eval` and `run` are used as synonyms for it. This code together with the skeleton of the `BachTSimul` class is presented in Figure 8.11.

8.1.5 Using the command-line interperter

The complete code is listed in appendix B. It is organized in four files, one for each of the three classes identified above together with one for the definition of the case classes. For the ease of use, they have been concatenated to form a single file, called `bacht-cli.scala`, following our aim to write a command-line interpreter for BachT.

Scala offers a very practical mechanism to write methods in a postfix form. As a result, we shall subsequently write `ag run "tell(t)"` instead of `ag.run("tell(t)")`. Thanks to this facility, the command-line interpreter can be used as illustrated in Figure 8.12 to run the agent of subsection 8.1.1. There, after having launched the scala interpreter, we load the file `bacht-cli.scala` and then evaluate the agent $(tell(t); get(u))||(get(t); tell(u))$, after which we empty the store by evaluating `bb reset`. As the reader will notice, this corresponds to what we aimed at in subsection 8.1.1. Note that, for the ease of reading, we decorate tuples with their number of occurrences instead of listing these occurrences in sequence.

As another example, we ask to the interpreter to evaluate the following expression:

$$(ask(t)\ ;\ tell(u))\ +\ ((nask(s); ask(t))\ ||\ (tell(t)\ ;\ get(t)))$$

```
def run_one(agent: Expr):(Boolean,Expr) = {

   agent match {

      case bacht_ast_primitive(prim,token) =>
         {  if (exec_primitive(prim,token)) { (true,bacht_ast_empty_agent()) }
            else { (false,agent) }
         }

      case bacht_ast_agent(";",ag_i,ag_ii) =>
         {  run_one(ag_i) match
               { case (false,_) => (false,agent)
                 case (true,bacht_ast_empty_agent()) => (true,ag_ii)
                 case (true,ag_cont) => (true,bacht_ast_agent(";",ag_cont,ag_ii))
               }
         }

      ...

   }
}


def exec_primitive(prim:String,token:String):Boolean = {
       prim match
         { case "tell" => bb.tell(token)
           case "ask"  => bb.ask(token)
           case "get"  => bb.get(token)
           case "nask" => bb.nask(token)
         }
   }

}
```

Figure 8.7: BachT-simulator: primitive and sequential composition

```
val bach_random_choice = new Random()

def run_one(agent: Expr):(Boolean,Expr) = {

   agent match {

      ...

      case bacht_ast_agent("||",ag_i,ag_ii) =>
         {  var branch_choice = bach_random_choice.nextInt(2)
            if (branch_choice == 0)
              { run_one( ag_i ) match
                   { case (false,_) =>
                       { run_one( ag_ii ) match
                           { case (false,_)
                               => (false,agent)
                             case (true,bacht_ast_empty_agent())
                               => (true,ag_i)
                             case (true,ag_cont)
                               => (true,bacht_ast_agent("||",ag_i,ag_cont))
                           }
                       }
                     case (true,bacht_ast_empty_agent())
                               => (true,ag_ii)
                     case (true,ag_cont)
                               => (true,bacht_ast_agent("||",ag_cont,ag_ii))
                   }
               }
            else
              { run_one( ag_ii ) match
                   { case (false,_) =>
                       { run_one( ag_i ) match
                           { case (false,_)
                               => (false,agent)
                             case (true,bacht_ast_empty_agent())
                               => (true,ag_ii)
                             case (true,ag_cont)
                               => (true,bacht_ast_agent("||",ag_cont,ag_ii))
                           }
                       }
                     case (true,bacht_ast_empty_agent())
                               => (true,ag_i)
                     case (true,ag_cont)
                               => (true,bacht_ast_agent("||",ag_i,ag_cont))
                   }
               }

      }
```

Figure 8.8: BachT-simulator: parallel composition

```
val bach_random_choice = new Random()

def run_one(agent: Expr):(Boolean,Expr) = {

   agent match {

      ...

      case bacht_ast_agent("+",ag_i,ag_ii) =>
         {  var branch_choice = bach_random_choice.nextInt(2)
            if (branch_choice == 0)
              { run_one( ag_i ) match
                   { case (false,_) =>
                       { run_one( ag_ii ) match
                           { case (false,_) => (false,agent)
                             case (true,bacht_ast_empty_agent())
                               => (true,bacht_ast_empty_agent())
                             case (true,ag_cont)
                               => (true,ag_cont)
                           }
                       }
                     case (true,bacht_ast_empty_agent())
                       => (true,bacht_ast_empty_agent())
                     case (true,ag_cont)
                       => (true,ag_cont)
                   }
               }
            else
              { run_one( ag_ii ) match
                   { case (false,_) =>
                       { run_one( ag_i ) match
                           { case (false,_)
                               => (false,agent)
                             case (true,bacht_ast_empty_agent())
                               => (true,bacht_ast_empty_agent())
                             case (true,ag_cont)
                               => (true,ag_cont)
                           }
                       }
                     case (true,bacht_ast_empty_agent())
                       => (true,bacht_ast_empty_agent())
                     case (true,ag_cont)
                       => (true,ag_cont)
                   }
               }
         }
```

Figure 8.9: BachT-simulator: non-deterministic choice

```
def bacht_exec_all(agent: Expr):Boolean = {

    var failure = false
    var c_agent = agent
    while ( c_agent != bacht_ast_empty_agent() && !failure ) {
       failure = run_one(c_agent) match
            { case (false,_)          ⇒ true
              case (true,new_agent)  ⇒
                 { c_agent = new_agent
                   false
                 }
            }
        bb.print_store
        println("\n")
    }

    if (c_agent == bacht_ast_empty_agent()) {
       println("Success\n")
       true }
    else {
       println("Failure\n")
       false }

}
```

Figure 8.10: BachT-simulator: main loop

```
import scala.util.Random
import language.postfixOps

class BachTSimul(var bb: BachTStore) {

   val bacht_random_choice = new Random()

   def run_one(agent: Expr):(Boolean,Expr) = { ... }

   def bacht_exec_all(agent: Expr):Boolean = { ... }

   def exec_primitive(prim:String,token:String):Boolean = { ... }

}


object ag extends BachTSimul(bb) {

  def apply(agent: String) {
     val agent_parsed = BachTSimulParser.parse_agent(agent)
     ag.bacht_exec_all(agent_parsed)
  }

  def eval(agent:String) { apply(agent) }
  def run(agent:String) { apply(agent) }

}
```

Figure 8.11: BachT-simulator: the `BachTSimul` class and the object `ag`

```
dda$scala
Welcome to Scala version 2.11.7 (OpenJDK 64-Bit Server VM, Java 1.6.0_24).
Type in expressions to have them evaluated.
Type :help for more information.

scala> :load bacht-cli.scala
Loading bacht-cli.scala ...
...

scala>ag run "(tell(t);get(u)) || (get(t);tell(u))"
{ t(1) }
{ }
{ u(1) }
{ }
Success

scala>bb reset

scala>
```

Figure 8.12: Running the BachT command line interperter

```
dda$scala
Welcome to Scala version 2.11.7 (OpenJDK 64-Bit Server VM, Java 1.6.0_24).
Type in expressions to have them evaluated.
Type :help for more information.

scala> :load bacht-cli.scala
Loading bacht-cli.scala ...
...

scala>ag run "(ask(t);tell(u)) + ((nask(s);ask(t)) || (tell(t);get(t)))"
{ }
{ t(1) }
{ t(1) }
{ }
Success

scala>bb reset

scala>
```

Figure 8.13: Running the BachT command line interperter

```
dda$scala
Welcome to Scala version 2.11.7 (OpenJDK 64-Bit Server VM, Java 1.6.0_24).
Type in expressions to have them evaluated.
Type :help for more information.

scala> :load bacht-cli.scala
Loading bacht-cli.scala ...
...

scala>ag run "(ask(t);tell(u)) + (nask(s);ask(t) || (tell(t);get(t))"
{ t(1) }
{ t(1) }
{ t(1) }
{ }
Success

scala>bb reset

scala>
```

Figure 8.14: Running the BachT command line interperter

In this example, the *ask(t)* in the first part of the choice cannot be executed, as the store is empty of any token. Only the second part of the choice can be executed. This can provide two different results. Indeed, in the parallel composition, the left part as well as the right part can start: *nask(s)* can be successful as well as *tell(t)*. If the computation starts with the left part, *nask(s)*, then *ask(t)* must wait for *tell(t)* to deposit a token t on the store. Before the execution of *get(t)*, the procedure verifies if there is a pending primitive for t, which is the case with *ask(t)*. The *ask(t)* primitive being executed, *get(t)* can be invoked to retrieve t. This produces the first trace of execution in Figure 8.13.

If the computation starts with the right part of the parallel composition, a token is first placed on the store. Before executing *get(t)*, the primitive *nask(s)* checks for the absence of s, then *ask(t)* checks for the presence of t, and finally, *get(t)* retrieves the token t from the store. This produces the second trace of execution in Figure 8.14.

As a third example more related to the common life, let us consider the following situation of a holydaymaker that hesitates between two destinations for his next holidays: to the Canary Islands or to some Mountains in France. His final choice is dictated by the local weather conditions, namely by the confirmation of a sunny sky in the islands, or by a high fresh snowfall in the mountains. Assume the two conditions are respectively represented by a token s and

a token f on the store consulted by the holidaymakers. Moreover let the two possibilities for his choices be represented by tokens c and m. With these tokens, the questioning of the store state about the weather conditions is done with the primitives *ask(s)* and *ask(f)*. Then the final decision is represented by the following process:

$$ask(s)\ ;\ tell(c)\ +\ ask(f)\ ;\ tell(m)$$

8.2 A command line simulator for BachT

8.2.1 Introduction

Our BachT interpreter provides simulations of the execution of a BachT agent with as result successful or failure. However, it does not allow for real parallel executions or real competition of alternative in a choice. Moreover, once blocked a computation cannot be awakened by the concurrent execution of another agent.

The command line simulator we shall design in this section allows to circumvent these difficulties. Essentially, we shall (implictly) use threads to launch parallel execution and, on the way, we shall directly code our REPL (read-eval-processing-loop) to offer the user a more direct way of entering his agents. Before going into the technicalities, let us first show the interface we would like to use.

```
 Welcome to BachT version 1.
 Type in agents to evaluate them.

BachT> tell(t).
BachT> >> Request 1 launched

BachT> >> tell(t) successfully terminated
       >> store : { t(1) }

BachT> >> Request 1 successfully terminated
```

We thus want to provide the user with a specific environment for BachT, in which he introduces directly the agents that are to be executed. A request is associated with every agent. In case of a succesful execution, a specific answer is printed, together with the new state of the store. If the state of the store does not permit the execution of the agent, a request is launched, but no result is produced. This situation is shown in Figure 8.15. In this figure, a request for the presence of a token u is launched. As the initial store is empty, the request cannot be

```
 Welcome to BachT version 1.
 Type in agents to evaluate them.

BachT> ask(u).
BachT> >> Request 1 launched

BachT>
```

Figure 8.15: The BachT simulator in command line with a waiting request

executed, and is placed in a waiting state. However a new prompt *BachT* is then printed, for the introduction of a new agent.

In case a new agent *tell(u)* is launched, it generates a second request, that can be successfully executed. The introduction of a token u on the store provides the condition for the first waiting request to be executed. Both requests are then executed, with an indication of the state of the store. Figure 8.16 shows this situation.

Let us suppose we want now to execute the following choice compositional agent *ask(u);tell(u) + get(t);tell(t)*, starting from an empty store. This agent is composed of two sequential subagents. Every subagent begins with a primitive than cannot be executed, as their conditions of execution are not fulfilled by the store. Moreover the operator being a choice, the agent will be completely executed if one of the subagent can be executed. The choice between these two possibilities will be based on the first step of execution. If the conditions on the store change in such a way that the first step *ask(u)* can be executed, then the first subagent will be chosen. On the contrary if a new state of the store permits to execute the first step *get(t)*, then the second subagent is chosen. Figure 8.17 shows this situation, with a modification of the store that permits the second subagent to execute.

Let us now present the different parts of the command line simulator. It is composed of four main components : a parser of agents, an executable class, a representation of the store and finally an object that performs the execution of a BachT agent. Identically to the interpreter, the structure of a parsed agent is represented in internal structures, as case classes of an abstract class called `Expr`. Figure 8.18 presents the abstract class `Expr` and the three case classes corresponding to respectively an empty agent, a primitive, and a composed agent.

8.2.2 The parser

The parser used for the parsing of BachT agents is the same as the one used for the interpreter. It presents itself as a class `BachTParsers` inheriting the class `RegexParsers` useful for

```
 Welcome to BachT version 1.
 Type in agents to evaluate them.

BachT> ask(u).
BachT> >> Request 1 launched

BachT> tell(u).
BachT> >> Request 2 launched

BachT> >> tell(u) successfully terminated
       >> store : { u(1) }

BachT> >> ask(u) successfully terminated
       >> store : { u(1) }

BachT> >> Request 1 successfully terminated

BachT> >> Request 2 successfully terminated

BachT>
```

Figure 8.16: The BachT simulator in command line with a second request liberating the first one

the parsing of regular expressions. For every primitive a case class is returned. Concerning the priorities of the operators, the sequential composition binds more than the parallel composition, which itself binds more than the non-determinstic choice operator. This order is reflected in the definition order of the functions responsible of the parsing of an agent : `compositionChoice`, `compositionPara`, `compositionSeq`, `simpleAgent` and `parenthesizedAgent`. An object instantiating the `BachTParsers` is also defined, with two methods to parse primitives and compositionnally composed agents. The codes of the parser an the object are available in annex (see chapter B in section B.2.1).

8.2.3 Executing agents

The `B_exec` class is in charge of the execution of a BachT agent. Starting from the parsed agent, the class constructs a list of pairs expression-expression (*Expr*,*Expr*). The first element of the pair is the first step to be executed, the second one is the continuation of this execution. The function `ag_first_steps` constructs the list according to the nature of the parsed agent, i.e a primitive, a choice agent, a sequential one, or a parallel one. Figure 8.19 shows its code.

For a primitive `B_AST_Primitive(b_prim, token)` the list is obviously composed of the

```
 Welcome to BachT version 1.
 Type in agents to evaluate them.

BachT> ask(u);tell(u) + get(t);tell(t).
BachT> >> Request 1 launched

BachT> tell(t).
BachT> >> Request 2 launched

BachT> >> tell(t) successfully terminated
       >> store : { t(1) }

BachT> >> get(t) successfully terminated
       >> store : { t(0) }

BachT> >> tell(t) successfully terminated
       >> store : { t(1) }

BachT> >> Request 1 successfully terminated

BachT> >> Request 2 successfully terminated

BachT>
```

Figure 8.17: The BachT simulator in command line with a choice between two subagents

```
class Expr
case class B_AST_Empty_Agent() extends Expr
case class B_AST_Primitive(primitive: String, token: String) extends Expr
case class B_AST_Agent(op: String, primitive: Expr, agent: Expr) extends Expr
```

Figure 8.18: The abstract BachT data class

primitive followed by the empty agent. For a choice agent `B_AST_Agent("+",ag_i,ag_ii)`, the list is constructed by a recursive call of the function `ag_first_steps` on the first agent `ag_i` concatenated with the same recursive call on the second agent `ag_ii`. In case of a sequential agent `B_AST_Agent(";",ag_i,ag_ii)`, the list is composed of the first step of the first agent `ag_i`, followed as second element by the continuation of the first agent in sequential composition with the second agent `ag_ii`. Finally for a parallel composition `B_AST_Agent("||",ag_i,ag_ii)`, the list is obtained in the same way as for the sequential composition but here for both agents `ag_i` and `ag_ii`.

The processing of a choice composition implies that only one of the subagent that is part of the choice will be executed. The list of pairs of expressions obtained with the `ag_first_steps` function is decomposed in two vectors. The first one contains the continuations of the first steps, and the second one contains the pairs of first steps, associated with an integer that represents the index of its associated continuation, in the vector of continuations. These vectors are the results of two functions : `vect_ag_first_steps` and `linedx_ag_first_steps`. Their codes are available in annex (see chapter B in section B.2.2).

When a choice agent has to be executed, the vector of continuations and the list of first steps-index are constructed. The list is randomly permuted with a shuffle instruction, and the result is transferd to a function `exec_l_choice` for execution. The is done by the following piece of code.

```
case B_AST_Agent("+",ag_i,ag_ii) => {
 var lstEE = ag_first_steps(B_AST_Agent("+",ag_i,ag_ii))
 var lstEV = vect_ag_first_steps(lstEE)
 var lstEI = lindex_ag_first_steps(lstEE,0)
 var i = exec_l_choice(Random.shuffle(lstEI))
 exec(lstEV(i))
 true
}
```

This code is part of an `exec` function, that executes the agents following their nature, i.e a primitive, a sequential composition, a parallel composition or a choice composition. Figure 8.20 shows the complete code of the `exec` funtion.

In case of a primitive, the function invokes a `exec_primitive` function. This function uses the functions associated with the primitive of the BachT language, i.e. *tell*, *ask*, *nask* and *get*. These functions are defined in the store description, in section 8.2.4. Figure 8.21 shows the code of this function. Following the nature of the primitive, it invokes a corresponding function defined in the store.

For a sequential composition, the `exec` function is recursively called for both agent `ag_i` and `ag_ii`. This is done by the following part of the code of the `exec` function.

```
case B_AST_Agent(";",ag_i,ag_ii) => {
               if (exec(ag_i)) { exec(ag_ii) } else { false}
}
```

For a parallel composition, threads are associated with the execution to both agents part of the composition. This is done by the following part of the code of the `exec` function.

```
case B_AST_Agent("||",ag_i,ag_ii) => {
 val t1 = thread(exec(ag_i))
 val t2 = thread(exec(ag_ii))
 t1.join
 t2.join
true
```

8.2.4 The store

As for the interpreter, the store is implemented as a mutable map in Scala. Every token added on it is represented in the form of an association of this token with a number representing the number of its occurrences on the store. In order to reach a more flexible behaviour the four primitives *tell*, *ask*, *nask* and *get* are now associated with a thread. Every primitive that executes has thus to take a lock on the store through a synchronized declaration. Following the nature of the primitive, different behaviours are possible. A *tell* primitive is always succesful. After having taken the lock, the primitive simply adds the token on the store. Following that it is already present or not on the store, the primitive increases its number of occurrences by 1, or creates a new registration *(t,1)* into the map. Finally, the primitive notifies this new situation to other waiting processes. Figure 8.22 presents the code of the *tell* primitive.

```
def ag_first_steps(b_ag : Expr) : List[(Expr,Expr)] = {

b_ag match {

   // a primitive is the first step followed by the empty agent
   case B_AST_Primitive(b_prim,token) => {
(B_AST_Primitive(b_prim,token),B_AST_Empty_Agent())::Nil
   }

   // for choice agent, recursive call of the function for
   // every element of the choice
   case B_AST_Agent("+",ag_i,ag_ii) => {
ag_first_steps(ag_i):::ag_first_steps(ag_ii)
   }

   // a sequence distinguishes both parts of Expression-Expression
   case B_AST_Agent(";",ag_i,ag_ii) => {
continuation(ag_first_steps(ag_i),ag_ii,";")
   }

   // for parallel agent, ag_i with its continuity in parallel with
   // ag_ii, and vice versa
   case B_AST_Agent("||",ag_i,ag_ii) => {
continuation(ag_first_steps(ag_i),ag_ii,"||"):::
                  continuation(ag_first_steps(ag_ii),ag_i,"||")
   }

}

}
```

Figure 8.19: Command line simulator : the construction of the list of first steps followed by their continuation

```
def exec(b_ag_parsed : Expr) : Boolean = {
b_ag_parsed match {
case B_AST_Empty_Agent() => {true}
case B_AST_Primitive(b_prim,token) => {
exec_primitive(b_prim,token);
 }
case B_AST_Agent(";",ag_i,ag_ii) => { if (exec(ag_i)) { exec(ag_ii) } else { false} }
case B_AST_Agent("||",ag_i,ag_ii) => {
                 val t1 = thread(exec(ag_i))
                 val t2 = thread(exec(ag_ii))
                 t1.join
                 t2.join
true
}
case B_AST_Agent("+",ag_i,ag_ii) => {
var lstEE = ag_first_steps(B_AST_Agent("+",ag_i,ag_ii))
var lstEV = vect_ag_first_steps(lstEE)
var lstEI = lindex_ag_first_steps(lstEE,0)
var i = exec_l_choice(Random.shuffle(lstEI))
exec(lstEV(i))
true
}
}
}
```

Figure 8.20: Command line simulator : the `exec` function of a parsed agent

```
def exec_primitive(b_prim : String, token : String) = {
b_prim match
{ case "tell" => bb.tell(token)
  case "ask"  => bb.ask(token)
  case "get"  => bb.get(token)
  case "nask" => bb.nask(token)
}
}
```

Figure 8.21: Command line simulator : the `exec_primitive` function

```
def tell(str : String) = bb.synchronized {
if(mapTok contains str) {
mapTok(str) = mapTok(str) + 1
} else {
mapTok = mapTok ++ Map(str -> 1)
}
println(">> tell("+str+") successfully terminated")
print("       >> store :")
print_store
println()
print("BachT> ")
bb.notifyAll()
true
}
```

Figure 8.22: Command line simulator : the *tell* primitive

The *get* primitive has a similar behaviour to the *tell* primitive, except that the token to be retrieved from the store must be present on it with an occurrence of at least one. If this condition is not reached, the function will wait until a modification appears on the store that makes the desired tokens appear in the map representing the store (test mapTok contains str in the following piece of code). If the condition is met the *get* primitive reduced the number of occurrences of the token to be retrieved by one. Then as for the *tell* primitive, the *get* primitive notifies this new situation of the store. The code of the *get* primitive is as follows:

```
def get(str : String) = bb.synchronized {
while(!(mapTok contains str) || mapTok(str) == 0) {bb.wait()}
println(">> get("+str+") successfully terminated")
mapTok(str) = mapTok(str) - 1
print("       >> store :")
print_store
println()
print("BachT> ")
bb.notifyAll()
true
    }
```

Differently from the *tell* and *get* primitives, the *ask* and *nask* primitives do not modify the state of the store, as they only respectively check for the presence or absence of a specific token. There is then no need for them to notify to every waiting primitive the end of their action. But similarly to the *get* primitive, their execution depends on the state of the store. In particular in case of the *ask*, it is necessary for the requested token to be present on the store with at least one unit. In the case of the *nask* primitive, the condition is that no occurrence of the token is present on the store, or with a number of occurrences equal to zero. If their respective conditions are not met, the execution of these primitives will be suspended, until the action of another agent modifies the state of the store in such a way that meets their respective request. Figures 8.23 and 8.24 depict the code of these two primitives.

```
def ask(str : String) = bb.synchronized {
while(!(mapTok contains str) || mapTok(str) == 0) {bb.wait()}
  println(">> ask("+str+") successfully terminated")
print("       >> store :")
print_store
println()
print("BachT> ")
true
    }
```

Figure 8.23: Command line simulator : the `ask` primitive

```
def nask(str : String) = bb.synchronized {
while((mapTok contains str)) {bb.wait()}
  if(!(mapTok contains str) || mapTok(str) == 0) {
println(">> nask("+str+") successfully terminated")
println("       >> " + "token not present ")
print("       >> store :")
print_store
println()
print("BachT> ")
}
true
}
```

Figure 8.24: Command line simulator : the `nask` primitive

The BachT interpreter manages the choice or parallel composition of two agents by attributing randomly a 0 or 1 value to a `branch_choice` variable. Following this value, the evaluation starts with the first or the second subagent. In the command line simulator, the random choice is generalized by constituing a list with the different subagents that are part of the compositional choice. Only the first step of the subagent is tested as executable or not. Four boolean functions are in charge to test if the primitive constituing the first step is excecutable or not. They are respectively `test_tell`, `test_get`, `test_ask` and `test_nask`. The content of their code is the same as those developed for the threads. The only difference resides in the nature of the returned response, which is now a boolean. Figure 8.25 shows for instance the code of the `test_tell` function. Running a choice is done with the function `run_l_choice`. The code of this function is listed in Figure 8.26.

```
def test_tell(str : String) : Boolean = {
if(mapTok contains str) {
mapTok(str) = mapTok(str) + 1
} else {
mapTok = mapTok ++ Map(str -> 1)
}
println(">> tell("+str+") successfully terminated")
print("       >> store :")
print_store
println()
print("BachT> ")
true
}
```

Figure 8.25: Command line simulator : the `Test_tell` primitive

```
def run_l_choice(lst : List[(Expr,Int)]) : Int = bb.synchronized {
var r = l_choice(lst)
while(! r._1 ) { // if no executable first step found in the list
bb.wait() // wait
r = l_choice(lst) // after a notify, restart the search
}
return r._2 // if first step found, return the associate integer
}
```

Figure 8.26: command line simulator : the `run_l_choice` function

A thread is associated with the function. The action of the function consists in receiving the results of the tests of execution of the first steps, through the result of a function `l_choice`. If no executable first step is found in the list, the function will wait until a notify occurs, that relaunches a new research. When an executable step is found, an integer is returned. This integer represents the index in a table of the rest of the subagent for its execution.

8.2.5 The main object

The object `MYSIMINLINE` contains the main function of the command line simulator. It contains a loop that processes the different instructions introduced by the user. Figure 8.27 shows the

```
try {
 myag_parsed = BachTSimulParser.parse_agent(line)
 val mysimul = new B_Exec(myag_parsed)
 println("BachT> >> Request " + cpt + " launched")
 val t = mysimul.thread(mysimul.exec_gen(myag_parsed,cpt))
 bb.synchronized{
println()
print("BachT> ")
 }
 line = readLine()
}
```

Figure 8.27: Command line simulator : the main function for the execution of an agent

part of the code for the execution of a BachT agent. The agent is parsed and then a thread is associated with its execution.

The object proposes some other functions, like a `clear` function to make the store empty, or a `print` function to print its content. The function `history` is used to register up to the last five commands that were introduced. Finally the `halt` function permits to end the session.

The complete code of the command line simulator is listed in section B.2.3 of Chapter B.

8.2.6 Using the BachT Command Line Simulator

Let us now present some examples of the execution of the BachT command line simulator. Suppose we want the simulator to evaluate the following expression, starting from an empty store:

$$(ask(t)\ ;\ tell(u))\ +\ ((nask(s); ask(t))\ ||\ (tell(t)\ ;\ get(t)))$$

In this example, in the first subagent, the first step *ask(t)* cannot be executed. In the second subagent, both first primitives of the parallel composition are executable: the *nask(s)* as well as the *tell(t)*. The second subagent will then be selected for execution. Two threads are associated to both parts of the parallel composition. If the thread containing the *nask* primitive is executed first, the *nask(s)*will be successful, but the *ask(t)* will wait until a token t is introduced on the store. The second thread beginning with the *tell(t)* will start. The introduction of this token t will be notified, and will permit to the waiting *ask* to execute successfully. Finaly the *get(t)*

```
 Welcome to BachT version 1.
 Type in agents to evaluate them.

BachT> (ask(t);tell(u)) + (nask(s);ask(t) || tell(t);get(t)).
BachT> >> Request 1 launched

BachT> >> nask(s) successfully terminated
       >> token not present
       >> store : { }

BachT> >> tell(t) successfully terminated
       >> store : { t(1) }

BachT> >> ask(t) successfully terminated
       >> store : { t(1) }

BachT> >> get(t) successfully terminated
       >> store : { t(0) }

BachT> >> Request 1 successfully terminated

BachT>
```

Figure 8.28: Running the BachT command line simulator

executes and retrieves the token t from the store, producing a final empty store. The trace of this execution is presented in Figure 8.28.

If the thread associated with the second subagent of the parallel composition begins first, then both the *tell(t)* and *get(t)* primitives will be successfully executed. In the thread associated with the first subagent, the *nask(s)* will be successfully executed. But the *ask(t)* will face a store without any token t, resulting in a waiting state. Figure 8.29 shows the trace of execution of this second possibility. The *history* command is used to recall the agent $(ask(t)\ ;\ tell(u))\ +\ ((nask(s); ask(t))\ ||\ (tell(t)\ ;\ get(t)))$. A request numbered 2 is associated with the thread and launched. As the *ask(t)* is waiting, the request is not indicated as successfully terminated.

The *ask(t)* primitive will wait until the introduction of a token t on the store is notified by a new agent *tell(t)*. This wakes up the thread and permits it to successfully finish. Figure 8.30 shows this final trace of execution. A request numbered 3 is launched with the new agent *tell(t)*. It has for consequence to make request number 2 successfull before finishing itself successfully.

```
BachT> history.
history

     ! :(ask(t);tell(u)) + (nask(s);ask(t) || tell(t);get(t))
    !! :m4
   !!! :m3
    !v :m2
     v :m1

BachT> !

BachT> history mode : (ask(t);tell(u)) + (nask(s);ask(t) || tell(t);get(t))
BachT>> execute (y/n) : y

BachT> >> Request 2 launched

BachT> >> nask(s) successfully terminated
       >> token not present
       >> store : { t(0) }

BachT> >> tell(t) successfully terminated
       >> store : { t(1) }

BachT> >> get(t) successfully terminated
       >> store : { t(0) }

BachT>
```

Figure 8.29: Running the BachT command line simulator

8.3 A command-line interpreter for Dense Bach

8.3.1 Introduction

The primitives of Dense Bach being of the same nature of those of BachT, with the difference of manipulating several occurrences of token atomically, it is easy to extend the interpreter of section 8.1 to Dense Bach. As a result, our command-line interpreter is based on the same four components : a file of abstract data, a parser of agents, the implementation of the store, and a simulator to perform the execution of a Dense Bach agent based on the execution of basic Dense Bach primitives. As BachT and Dense Bach language definitions are very similar, with the only difference due to the presence of an added density to the tokens, a few adaptations of

```
BachT> tell(t).
BachT> >> Request 3 launched

BachT> >> tell(t) successfully terminated
       >> store : { t(1) }

BachT> >> ask(t) successfully terminated
       >> store : { t(1) }

BachT> >> Request 2 successfully terminated

BachT> >> Request 3 successfully terminated
```

Figure 8.30: Running the BachT command line simulator

```
class Expr
case class dbach_ast_empty_agent() extends Expr
case class dbach_ast_primitive(primitive: String, token: String,
                                        density: Int) extends Expr
case class dbach_ast_agent(op: String, agenti: Expr,
                                        agentii: Expr) extends Expr
```

Figure 8.31: The abstract Dense Bach data.scala file

the Scala code are only necessary.

The refinements of the `Expr` abstract class used to represent the structure of a parsed agent is again composed of three classes:

- a case class `bacht_ast_empty_agent` to represent the empty agent E,
- a case class `bacht_ast_primitive` to represent a primitive in the form of a triplet composed of a primitive type (tell, ask, nask, get), a token and its density.
- a case class `bacht_ast_agent` to represent a composed agent formed from an operator applied to two sub-agents `agenti` and `agentii`

Note however the presence of density in the second case. The code of the abstract Dense Bach data file is depicted in Figure 8.31.

8.3.2 The parser

The parser defines the class DenseBachParsers as inherited from the class RegexParsers. The two main differences with the parser developed for BachT lie on the one hand in the addition of a definition of a regular expression for the density and, on the other hand, in the definitions of the parsing of the primitives, that now take into account the presence of the density in the examined value. Technically this is achieved as follows. The notion of density is defined as a natural number:

```
def density   : Parser[Int] = ("[1-9][0-9]*").r ^^ {_.toInt}
```

Moreover, the definition of a primitive is refined as follows:

```
def primitive : Parser[Expr]   = "tell("~token~"("~density~"))" ^^ {
        case _ ~ vtoken ~ _ ~ vdensity ~ _ =>
                        dbach_ast_primitive("tell",vtoken,vdensity) }
```

The priorities of the three operators stay unchanged, stipulating that the sequential composition binds more than the parallel composition, which itself binds more than the non-deterministic choice operator. The definitions of the three functions of compositions takes into account the abstract classes as defined in the Dense Bach abstract data file. The code of the parser is presented in Figure 8.32.

Finally, the code of the parser provides also the definition of an object instanciating the `DenseBachParsers` for using it directly in the command-line interpreter. This is represented in the code of Figure 8.33.

8.3.3 The store

The implementation of the store has been adapted to take into account the density associated with a token. It still consists in a mapping associating a token with a number representing the number of its occurences on the store. The adaptation concerns the implementation of the four primitives `tell`, `get`, `ask` and `nask` in the class `DenseBachStore`. For instance, the execution of the `tell(t(n))` primitive checks again for the presence of the token `t` on the store. In case of a positive answer, it is now n instances of `t` that are added to the number of tokens `t` already present on the store. If there is no token `t` present on the store, then a new association `(t,n)` is added to the map. The `get(t(n))` primitive checks for the presence of `n` tokens `t` on the store. In case `t` is present with an occurence higher than `n`, then the `get` primitive decrements it by

```
class DenseBachParsers extends RegexParsers {

  def token      : Parser[String] = ("[a-z][0-9a-zA-Z_]*").r ^^ {_.toString}
  def density    : Parser[Int] = ("[1-9][0-9]*").r ^^ {_.toInt}

  val opChoice   : Parser[String] = "+"
  val opPara     : Parser[String] = "||"
  val opSeq      : Parser[String] = ";"

  def primitive : Parser[Expr]    = "tell("~token~"("~density~"))" ^^    {
        case _ ~ vtoken ~ _ ~ vdensity ~ _ =>
                              dbach_ast_primitive("tell",vtoken,vdensity)}
|
                                    "ask("~token~"("~density~"))" ^^ {
        case _ ~ vtoken ~ _ ~ vdensity ~ _ =>
                              dbach_ast_primitive("ask",vtoken,vdensity) }
|
                                    "get("~token~"("~density~"))" ^^ {
        case _ ~ vtoken ~ _ ~ vdensity ~ _ =>
                              dbach_ast_primitive("get",vtoken,vdensity) }
|
                                    "nask("~token~"("~density~"))" ^^ {
        case _ ~ vtoken ~ _ ~ vdensity ~ _ =>
                              dbach_ast_primitive("nask",vtoken,vdensity)}

  def agent = compositionChoice

  def compositionChoice : Parser[Expr] =
                               compositionPara~rep(opChoice~compositionChoice)
      ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => dbach_ast_agent(op,agi,agii) }

  def compositionPara : Parser[Expr] =
                              compositionSeq~rep(opPara~compositionPara) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => dbach_ast_agent(op,agi,agii) }

  def compositionSeq : Parser[Expr] =
                                    simpleAgent~rep(opSeq~compositionSeq) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => dbach_ast_agent(op,agi,agii) }

  def simpleAgent : Parser[Expr] = primitive | parenthesizedAgent

  def parenthesizedAgent : Parser[Expr] = "("~>agent<~")"
}
```

Figure 8.32: Parser: the class `DenseBachParsers`

```
object DenseBachSimulParser extends DenseBachParsers {

  def parse_primitive(prim: String) = parseAll(primitive,prim) match {
        case Success(result, _) => result
        case failure : NoSuccess => scala.sys.error(failure.msg)
  }

  def parse_agent(ag: String) = parseAll(agent,ag) match {
        case Success(result, _) => result
        case failure : NoSuccess => scala.sys.error(failure.msg)
  }

}
```

Figure 8.33: Parser : the object `DenseBachSimulParser`

`n`. The behaviour of the `ask(t(n))` primitive is similar to the one of the `get(t(n))` primitive concerning the check for the presence of n instances of `t`, but without retrieving any occurrence from the store. Finally the `nask(t(n))` primitive succeeds in case of a presence of `t` with a number strictly less than the density `n`, and fails in the other case. The code describing the class DenseBachStore is presented in Figure 8.34.

As for the implementation of the store for BachT, the code provides two auxiliary functions `print_store` and `clear_store`, for respectively printing the content of the store, and for resetting it to the empty map. Moreover an object `bb` of type `DenseBachStore`, with a function `reset` as a synonym for `clear_store` is also provided. These elements stay completely unchanged with regard to the code of the store as developped in BachT. They are represented in Figures 8.35 and 8.36.

8.3.4 The simulator

The simulator still consists in the implementation of a function `run_one`, that performs a transition step but now with respect to the operational semantics of section 4.1.2. The strategy of the implementation for the sequential, the parallel and the choice compositions stay unchanged with regard to the one developped for the BachT language. In particular for the choice or parallel operator, the first subagent to be executed is still selected on the base of a random value assigned to a `branch_choice` variable. Both subagents must be evaluated in case of the parallel composition, while the evaluation is limited to the first selected subagent for the choice composition. Function `dbach_exec_all` repeatedly executes `run_one` until the empty agent is reached or failure is produced. Because of the great similarity with the one of BachT, the code

```
import scala.collection.mutable.Map

class DenseBachStore {

   var theStore = Map[String,Int]()

   def tell(token:String,density:Int):Boolean = {
      if (theStore.contains(token))
        { theStore(token) = theStore(token) + density }
      else
        { theStore = theStore ++ Map(token -> density) }
      true
   }


   def ask(token:String,density:Int):Boolean = {
      if (theStore.contains(token))
             if (theStore(token) >= density) { true }
             else { false }
      else false
   }

   def get(token:String,density:Int):Boolean = {
      if (theStore.contains(token))
             if (theStore(token) >= density)
               { theStore(token) = theStore(token) - density
                 true
               }
             else { false }
      else false
   }

   def nask(token:String,density:Int):Boolean = {
      if (theStore.contains(token))
             if (theStore(token) >= density) { false }
             else { true }
      else true
   }


   ...
```

Figure 8.34: The `DenseBachStore class`

```
class DenseBachStore {

  ...

   def print_store {
      println("{")
      for ((t,d) <- theStore)
         println ( t + "(" + theStore(t) + ")" )
      println("}")
   }

   def clear_store {
      theStore = Map[String,Int]()
   }

}
```

Figure 8.35: The `DenseBachStore class` continued

```
object bb extends DenseBachStore {

   def reset { clear_store }

}
```

Figure 8.36: The `bb` object

```
Welcome to Scala version 2.11.7 (OpenJDK Server VM, Java 1.7.0_95).
Type in expressions to have them evaluated.
Type :help for more information.

scala> :load dbach-cli.scala
Loading dbach-cli.scala...
...

scala> ag run "(get(t(2));tell(u(3)))||(tell(t(3));ask(u(2)))"
{ t(3) }
{ t(1) }
{ t(1) u(3) }
{ t(1) u(3) }
Success
scala>
```

Figure 8.37: Running the Dense Bach command line interperter (1)

is not reproduced inside the main text of this thesis. It is however listed in appendix C.

8.3.5 Using the command-line interpreter

The full code of the Dense Bach interpreter is listed in appendix C. As for BachT, it is presented as a single file, called `dbach-cli.scala`, which is a concatenation of four files, three for the parser, the store and the simulator, and one for the definition of the case classes.

Using the facilities provided by Scala to write postfix notation, Figure 8.37 presents the result of the computation of the following expression:

$$(get(t(2)) \; ; tell(u(3))) \; || \; (tell(t(3)) \; ; ask(u(2)))$$

As second example, we propose to evaluate the following expression:

$$(ask(t(1)) \; ; tell(u(3))) \; + \; (nask(s(2)) \; ; get(t(2)) \; || \; (tell(t(3)) \; ; tell(s(1))))$$

The left part in the choice agent cannot be executed, due to the absence of any token `t` on the store. The right part being a parallel composition, the computation can start randomly with either `nask(s(2))` or `tell(t(3))`. In the present case `nask(s(2))` is the chosen first step, that succeeds as the store is empty, and leaves it unchanged. As there is no token `t` on the store, `get(t(2))` cannot be executed. This is then `tell(t(3))` that is executed, enriching the

```
Welcome to Scala version 2.11.7 (OpenJDK Server VM, Java 1.7.0_95).
Type in expressions to have them evaluated.
Type :help for more information.

scala> :load dbach-cli.scala
Loading dbach-cli.scala...
...

scala> ag run "(ask(t(1));tell(u(3))) +
                    (nask(s(2));get(t(2)) || tell(t(3));tell(s(1)))"
{ }
{ t(3) }
{ t(3) s(1) }
{ t(1) s(1) }
Success
```

Figure 8.38: Running the Dense Bach command line interpreter (2)

store with three tokens `t`. The next step can now be done by either `get(t(2))` or `tell(s(1))`. In this example, the choice has been made for `tell(s(1))`, enriching the store with one token `s`. Finally, `get(t(2))` is executed, retrieving two tokens `t` from the store. The total result of the execution is available in Figure 8.38.

8.4 A Command Line Simulator for Dense Bach

8.4.1 Introduction

Similarly to what we did for BachT, we have developed a command line simulator for the Dense Bach language. The primitives of Dense Bach differing from those of BachT only by the atomic manipulation of several occurences of tokens, the code of this Dense Bach command line simulator presents a few differences with the one for BachT. The global structure thus stays the same: the definition of an abstract data, a parser of agents, a class managing the execution of agents, an implementation of the store, and a global object containing the main method.

The refinement of the `Expr` abstract class used to represent the structure of a parsed agent is again composed of three classes:

- a case class `DB_AST_Empty_Agent` to represent the empty agent E,
- a case class `DB_AST_Primitive` to represent a primitive in the form of a triplet composed of a primitive type (tell, ask, nask, get), a token and its density.

```
class Expr
case class DB_AST_Empty_Agent() extends Expr
case class DB_AST_Primitive(primitive: String,
                        token: String, density: Int) extends Expr
case class DB_AST_Agent(op: String, primitive: Expr,
                        agent: Expr) extends Expr
```

Figure 8.39: The abstract Dense Bach data class

- a case class `DB_AST_Agent` to represent a composed agent formed from an operator applied to two sub-agents `agenti` and `agentii`

The code of the abstract Dense Bach data class is provided in annex (see chapter C.2.6 in section C.2.1).

8.4.2 The parser

As in the case of BachT, the class DenseBachParsers inherits from the class `RegexParsers`. The introduction of the density has for consequences on the one hand the need to define a regular expression for it, and on the other hand, to adapt the definitions of the parsing of the primitives, to take into account this density. These adaptations are exactly the same as those introduced in the interpreter. The order of priority of the three operators stay unchanged : firstly the sequentiality, then the parallelism and finally the choice. The definitions of the compositions are based on the abstract classes of the Dense Bach data class. An object instanciating the `DenseBachParser` is also available. It provides two functions for respectively parsing a primitive and parsing an agent. The complete code of the parser is listed in annex (see chapter C.2.6 in section C.2.2).

8.4.3 Executing agents

As for the BachT command line simulator, the DB-exec class is in charge of the execution of the Dense Bach agents. Receiving a parsed agent, an `exec` function executes it according to its nature : a primitive, a sequentialy composed agent, an agent resulting from a parallel composition or finally from a choice composition. For the primitive, a specific `exec_primitive` function processes the parsed agent, with the following code:

```
case DB_AST_Primitive(db_prim,token,density) => {
```

```
def exec_primitive(db_prim : String, token : String, density : Int) = {
db_prim match
{ case "tell" => bb.tell(token,density)
  case "ask"  => bb.ask(token,density)
  case "get"  => bb.get(token,density)
  case "nask" => bb.nask(token,density)
}
}
```

Figure 8.40: Command line simulator : the `exec_primitive` function

```
 exec_primitive(db_prim,token,density);
}
```

The function `exec_primitive` invokes functions defined for the execution of the *tell*, the *get*, *ask* or the *nask*. These functions are used for the management of the store and are presented in section 8.4.4. Figure 8.41 presents the code of the `exec_primitive` function.

In case of a parsed agent resulting from a parallel composition, threads are associated with the execution of both agents *ag_i* and *ag_ii* that are part of the composition, following this code:

```
case DB_AST_Agent("||",ag_i,ag_ii) => {
 val t1 = thread(exec(ag_i))
 val t2 = thread(exec(ag_ii))
 t1.join
 t2.join
 true
}
```

For the execution to be completed, both threads must be joined at the end of their executions.

For a sequential composition, the code proceeds with the execution of the first agent *ag_i*. When completed, it continues with the execution of the second agent *ag_ii*. The code is the following:

```
case DB_AST_Agent(";",ag_i,ag_ii) =>
                    { if (exec(ag_i)) { exec(ag_ii) } else { false} }
```

For a choice composition, only one of the subagents that are parts of the choice will be executed. The selection of an executable subagent is based on the first step. Every subagent is

presented as a first step primitive, followed by a continuation. These pairs (first step, continuation) are grouped in a list constructed by a function `ag_first_steps`. To ease its manipulation, this list is decomposed in two lists: one storing in a vector the previous continuations, and another one of the pair constitued by the first steps, and the index in the vector pointing to the associated continuation. This second list is randomly permutated and given as parameter to a function `exec_l_choice`. This function returns the first subagent that can be executed based on its first step. The code of the processing of a choice agent is as followed :

```
case DB_AST_Agent("+",ag_i,ag_ii) => {
 var lstEE = ag_first_steps(DB_AST_Agent("+",ag_i,ag_ii))
 var lstEV = vect_ag_first_steps(lstEE)
 var lstEI = lindex_ag_first_steps(lstEE,0)
 var i = exec_l_choice(Random.shuffle(lstEI))
 exec(lstEV(i))
 true
}
```

The complete code of the `exec` function is given in annex (see chapter C.2.6 in section C.2.4).

```
def exec(db_ag_parsed : Expr) : Boolean = {
db_ag_parsed match {
case DB_AST_Empty_Agent() => {true}
case DB_AST_Primitive(db_prim,token,density) => {
exec_primitive(db_prim,token,density);
 }
case DB_AST_Agent(";",ag_i,ag_ii) => { if (exec(ag_i))
                              { exec(ag_ii) } else { false} }
case DB_AST_Agent("||",ag_i,ag_ii) => {
                 val t1 = thread(exec(ag_i))
                 val t2 = thread(exec(ag_ii))
                 t1.join
                 t2.join
true
}
case DB_AST_Agent("+",ag_i,ag_ii) => {
var lstEE = ag_first_steps(DB_AST_Agent("+",ag_i,ag_ii))
var lstEV = vect_ag_first_steps(lstEE)
var lstEI = lindex_ag_first_steps(lstEE,0)
var i = exec_l_choice(Random.shuffle(lstEI))
exec(lstEV(i))
true
}
}
}
```

Figure 8.41: Command line simulator : the `exec_primitive` function

The code of the function `exec_l_choice` invokes a function `l_choice` defined in the store, that we will now investigate.

8.4.4 The store

The store is again constitued of a mapping associating a token with a number representing the number of its occurrences on the store. Four functions corresponding to the four primitives `tell`, `get`, `ask` and `nask` are defined. In a logic of threads, every primitive has to take the lock of the store. The `tell(t(n))` primitive checks for the presence of a token `t` on the store. If not present, a new association `(t,n)` is added to the map. If present, the number `n` is added to the

number of tokens `t` already present on the store. As the action of the `tell` primitive modifies the store, it has to be notified. The `get(t(n))` primitive checks for the presence of `n` tokens `t` on the store. If this token is present with an occurrence higher than `n`, it is decremented by `n`. If the token `t` is not present, or present with a density strictly less than `n`, the function will wait until a modification of the store is notified. If this modification permits to the `get(t(n))` to continue, it executes the retrieval of the `n` tokens. As this constitutes a modification of the store, this successful action is also notified. The `ask(t(n))` primitive is similar to the `get(t(n))` primitive, but executes no retrieval. At the end of the execution, no notification is needed as the content of the store is not modified. The `nask(t(n))` primitive is succesful if the number of token `t` on the store is strictly less then `n`. If not the primitive waits until a modification of the store is notified. If this modification acts as required by the `nask` primitive, it can perform fully its action. No notification is needed at the end of its execution. The code describing these primitives is presented in annex (see chapter C.2.6 in section C.2.3).

In the case of the execution of an agent formed by a choice, we use the selection of subagents presented in section 8.4.3. Every subagent is thus associated with its first step primitive, followed by a continuation. Every first step, that is a primitive, must be checked as executable or not. This is done by boolean test functions that check the same conditions for their execution as the four previous primitive functions `tell` , `get`, `ask` and `nask`. As they are selected once successful, they do more than a test and actually perform, if successfully selected, their action on the content of the store.

Every first step primitive is tested as executable or not, with the function `run_l_choice`. In case no first step is found executable, the process waits until a notified modification of the store restarts the analysis of all the first steps. In case of a successful first step, then the index of its corresponding continuation in the vector of continuation is returned. The following code is responsible of this execution.

```
def l_choice(lst : List[(Expr,Int)]) : Int = bb.synchronized {
var r = run_l_choice(lst)
while(! r._1 ) {
bb.wait()
r = run_l_choice(lst)
}
return r._2
}
```

8.4.5 The main object

As for the BachT command line simulator, a main object called `MYSimInLine` contains the main function that will execute a Dense Bach agent. The core of this function invokes the `DenseBachSimulParser` object to parse the agent. A thread is associated with the execution of the resulted parsing. For the ease of the presentation, a number is associated with every launched thread. The following code presents the core of the main function:

```
try {
myag_parsed = DenseBachSimulParser.parse_agent(line)
val mysimul = new DB_Exec(myag_parsed)
println("DBach> >> Request " + cpt + " launched")
    val t = mysimul.thread(mysimul.exec_gen(myag_parsed,cpt))
bb.synchronized{
println()
print("DBach> ")
}
line = readLine()
}
```

Other useful functionalities are added, as an `history` command that lists the last five commands, a `print` command that prints the content of the store, and a `clear` command that makes the store empty.

The complet code of the Dense Bach command line simulator is listed in section C.2.6 of appendix C.

8.4.6 Using the Dense Bach Command Line Simulator

In this section we present some examples of the execution of the Dense Bach command line simulator. Suppose we want the command line simulator to evaluate the following expression, starting from an empty store:

$$(ask(t(2)) \; ; tell(u(3))) \; + \; (nask(s(1)); ask(t(2))$$

In this example, in the first subagent, the first step *ask(t(2))* cannot be executed. In the second subagent, the first primitive *nask(s(1))* is executable. The second subagent in the choice is thus to be selected for an execution. Concretely a first thread is launched, associated with the agent $(ask(t(2)) \; ; tell(u(3))) \; + \; (nask(s(1)); ask(t(2))$. The *nask(s(1))* primitive is successfully

```
 Welcome to Dense Bach version 1.
 Type in agents to evaluate them.

DBach> ask(t(2));tell(u(3)) + nask(s(1));ask(t(2)).
DBach> >> Request 1 launched

DBach> >> nask(s(1)) successfully terminated
       >> token s not present
       >> store : { }

DBach>
```

Figure 8.42: Using the Dense Bach command line simulator

executed, but the *ask(t(2))* that follows in sequence is not executable, as the store is empty from any token t. The trace of this first execution is presented in Figure 8.42.

In order to make the *ask(t(2))* executable, we type in as a next request a *tell(t(3))* primitive, that introduces 3 tokens t on the store. A second request is indeed launched that places the 3 tokens t on the store. As the *tell(t(3))* notifies the modification of the store, the waiting *ask(t(2))* is resumed and finishes successfully. The trace of this second execution is presented in Figure 8.43.

As a second example, suppose we want to execute the following agent, starting from an empty store.

$$(nask(s(1)) \; ; \; get(t(1))) \; || \; (ask(u(2)) \; ; \; tell(t(2)) \; + \; tell(t(2)) \; ; \; tell(t(3))).$$

A first request is launched for the parallel composition. A thread is associated with every both subagent: the sequential composition of $nask(s(1)) \; ; \; get(t(1))$ on the one hand, and with the choice composition $ask(u(2)) \; ; \; tell(t(2)) \; + \; tell(t(2)) \; ; \; tell(t(3))$ on the other hand. In the first subagent, the *nask(s(1))* produces a successful result. The *get(t(1))* cannot be executed, as the store does not contain any token t. In the second subagent, no alternative of the choice can be executed, as the *ask(u(2))* and the *ask(s(2))* primitives cannot be executed. Figure 8.44 shows the trace of this first execution.

```
 Welcome to Dense Bach version 1.
 Type in agents to evaluate them.

DBach> ask(t(2));tell(u(3)) + nask(s(1));ask(t(2)).
DBach> >> Request 1 launched

DBach> >> nask(s(1)) successfully terminated
       >> token s not present
       >> store : { }

DBach> tell(t(3)).
DBach> >> Request 2 launched

DBach> >> tell(t(3)) successfully terminated
       >> store : { t(3) }

DBach> >> ask(t(2)) successfully terminated
       >> store : { t(3) }

DBach> >> Request 1 successfully terminated

DBach> >> Request 2 successfully terminated

DBach>
```

Figure 8.43: Using the Dense Bach command line simulator

```
 Welcome to Dense Bach version 1.
 Type in agents to evaluate them.

DBach> nask(s(1));get(t(1)) || (ask(u(2));tell(t(2)) + ask(s(2));tell(t(3))).
DBach> >> Request 1 launched

DBach> >> nask(s(1)) successfully terminated
       >> token s not present
       >> store : { }

DBach>
```

Figure 8.44: Using the Dense Bach command line simulator

Let us imagine we introduce a new agent *tell(s(4))*, corresponding to a second request. It will be successful, and the resulting insertion of 4 tokens s liberates the waiting thread associated

with the second subagent of the parallel composition. In particular the *ask(s(2))* primitive in the sequential composition $ask(s(2))$; $tell(t(3))$ is now executable. As the *tell(t(3))* primitive is also successfully executable, the second subagent of the choice composition is completly executed. Moreover the introduction of 3 tokens t permits to execute the *get(t(1))* primitive in the first request. Figure 8.45 shows the trace of this last execution.

```
 Welcome to Dense Bach version 1.
 Type in agents to evaluate them.

DBach> nask(s(1));get(t(1)) || (ask(u(2));tell(t(2)) + ask(s(2));tell(t(3))).
DBach> >> Request 1 launched

DBach> >> nask(s(1)) successfully terminated
       >> token s not present
       >> store : { }

DBach> tell(s(4)).
DBach> >> Request 2 launched

DBach> >> tell(s(4)) successfully terminated
       >> store : { s(4) }

DBach> >> ask(s(2)) successfully terminated
       >> store : { s(4) }

DBach> >> tell(t(3)) successfully terminated
       >> store : { t(3) s(4) }

DBach> >> get(t(1)) successfully terminated
       >> store : { t(2) s(4) }

DBach> >> Request 2 successfully terminated

DBach> >> Request 1 successfully terminated

DBach>
```

Figure 8.45: Using the Dense Bach command line simulator

8.5 Conclusion

In this chapter, we have proposed an interpreter and a command line simulator for BachT and Dense Bach. Our goal in providing them was to provide the reader with interpreters and simulators for executing agents in BachT and Dense Bach and thus to experience with them.

Following [BJ93], we have implemented the store as a token-indexed list. However, our implementation has been made simpler due to the fact that we first aimed at providing interpreters. In contrast, in [BJ93], Bosschere and Jacquet provide a real implementation based on threads for parallel executions and on semaphores associated with functors of logic terms in order to suitably handle mutual access to shared elements. As evidenced by the simulators, these design choices are however compatible with our implementation design. Indeed we have used threads but have kept the design simpler by using a global lock on the store and not one for each token. As noted in [BJ93], this is however a key factor which allows to generate speedups over our more naive implementation which requires to block the entire tuple space. It is also worth noting that, in contrast to [BJ93], we handle non-deterministic choice and thus provide an implementation support for a more elaborated process algebra.

LighTS [BCP07] is the core tuple space layer for the Lime middelware. Implemented in Java, it has an object-oriented design, permitting to provide dedicated constructs for the development of context-aware applications. Two packages constitue the core of Lights: a lights.interface one that contains the interfaces modeling the fundamental concepts of Linda, and the lights package that contains a built-in implementation of these interfaces. The goal of LighTS is to provide a simple and extensible implementation. We share this philosophy and even provide a simpler framework since we only deal with simple tokens and thus do not even provide interfaces. On the contrary, as for [BJ93], it is worth noting that we support the non-deterministic choice operator which is not supported by LighTS, and this without compromising the simplicity of our implementation.

TuCSoN is an infrastructure that offers services to enable the communication and coordination of distributed / concurrent independent software components called agents. Tuples centres constitute the core of TuCSoN in that they provide many features, in particular, the possibility to program reactions. The implementation is based on an implementation of Prolog in Java and is thus much more heavy than our implementation.

Finally, to design our implementation, we have taken inspiration of Pistache [MdM11], an implementation of the pi-Calculus as a Domain Specific Language in Scala. In particular, we have use similar case class structures and have handled parallel and choice compositions using

the same technique based on generating random numbers. However, the communication in BachT is completely different and has called for an implementation of the store and related primitives which is not present in Pistache.

Chapter 9

On the Implementation of Distributed Density

As for the BachT and Dense Bach languages in Chapter 8, we develop in this chapter an interpreter and a command line simulator for the Vectorized Dense Bach language introduced in Chapter 5. As this language shares properties with the MRT language, we proceed in the same way for MRT. Again these tools allow the reader to experiment with these languages and they argue for their implementability.

Section 9.1 presents the command-line interpreter for the Vectorized Dense Bach language. Section 9.2 presents the command line simulator for the same language. Sections 9.3 and 9.4 proceed similarly for the MRT language.

9.1 A command-line interpreter for Vectorized Dense Bach

9.1.1 Introduction

In comparison with the primitives of the Dense Bach language, those of the Vectorized Dense Bach language bring the new capacity to manipulate several occurrences of many different tokens atomically but keep the same nature. It is then easy to extend the interpreter developed for Dense Bach in Section 8.3 of Chapter 8 to the Vectorized Dense Bach language. We found basically the same four components : a file of abstract data, a parser of agents, the implementation of the store, and a simulator to perform the execution of a Vectorized Dense Bach agent based on the execution of the basic Vectorized Dense Bach primitives.

The Vectorized Dense Bach language permits to manipulate many instances of different tokens atomically. This implies that the structure of the data must be adapted to introduce the

notion of dense token, that can be recorded in a list. As a result the data structure presents a new abstract class *dtExpr*, which is refined in a class *dt*, with the two parameters *tok* of type string and *dens* of type integer, to represent a dense token. A second abstract class *vDBachExpr* represents (as for the BachT and Dense Bach languages) the structure of a parsed agent and is composed of three usual classes for the empty agent, for the primitives, and for the composed agent. Nevertheless the class representing a primitive manipulates a parameter of type list of dense tokens, as defined in the abstract class *dtExpr*. The code of these two abstract classes is as follows:

```
class dtExpr
case class dt(tok: String, dens: Int) extends dtExpr

class vDBachExpr
case class vdbach_ast_empty_agent() extends vDBachExpr
case class vdbach_ast_primitive(primitive: String,
                    lDenseToken: List[dt]) extends vDBachExpr
case class vdbach_ast_agent(op: String, agenti: vDBachExpr,
                      agentii: vDBachExpr) extends vDBachExpr
```

9.1.2 The parser

The parser defines the class *VDenseBachParsers* that extends the class *RegexParsers*. Two main differences with the parser developed for the Dense Bach language appear. Firstly two definitions of regular expressions are added to represent, on the one hand, a dense token, and on the other hand, a vector of dense tokens list. Secondly the definitions of the parsing of the primitives take into account these lists. Technically, the two definitions for a dense token, and a vector of them, are as follows:

```
def denseToken : Parser[dt] = token~"("~density~")" ^^ {
  case vtoken ~ _ ~ vdensity ~ _ => dt(vtoken,vdensity)
}
def vectDenseTokenList: Parser[List[dt]] = denseToken ~ rep("," ~ vectDenseTokenList)
    ^^ { case vdenseToken ~ List() => List(vdenseToken)
         case vdenseToken ~ List(op~lvdt) => List(vdenseToken):::lvdt
       }
```

The definition of a primitive, for instance the *tell* primitive, is as follows:

```
def primitive : Parser[vDBachExpr]   = "tell("~vectDenseTokenList~")" ^^ {
        case _ ~ vvectDenseTokenList ~ _
                    => vdbach_ast_primitive("tell",vvectDenseTokenList) }
```

The order of priorities between the three operators is maintained as it is in the Dense Bach parser. Finally, an object instanciating the *VDenseBachParsers* is provided for a direct use in the command-line interpreter. The complete code of the parser follows the developments of Chapter 8. As a result, it is not presented here but is available in section D.1.2 of Appendix D.

9.1.3 The store

The code implementing the store requires also some adaptation in order to manipulate lists of dense tokens. This concerns the implementations in the class *VDenseBachStore* of the four boolean functions describing the behaviour of the primitives *tell*, *ask*, *get* and *nask*. Concretely they are defined in an inductive way on the structure of the dense tokens lists. For instance for the *tell* primitive, the presence on the store of every dense token inside the list is checked. For a token that is already present on the store, its density is added to the density registered on the store. If not present, the token and its density are added to the map representing the store. The function is called recursively on the tail of the list, up to the moment it is empty, in which case the result is true. The code of the *tell* primitive is as follows:

```
def tell(vectDenseTokenList:List[dt]):Boolean = {
    vectDenseTokenList match {
        case Nil => true
        case dt(tok,dens)::l => {
            if (theStore.contains(tok)) {
                theStore(tok) = theStore(tok) + dens
                tell(l)
            } else {
                theStore = theStore ++ Map(tok -> dens)
                tell(l)
            }
        }
    }
}
```

The *ask* primitive proceeds in a similar way. The presence on the store of every dense token inside the list is checked, with a number at least equal to the density associated with the token in the list. If these two conditions are respected for every token in the list, the function returns true. Otherwise the first token for which the check fails – because it is not present on the store, or because it is present on it but without a sufficient number – will interrupt the process of verification, and the function fails. The code of the *get* function is the same as for the *ask* function, except that a successful check implies to reduce the number associated with

the dense token on the store, with its density expressed in the list. Any failure in the check process interrupts it and makes the function fails. Finally the *nask* function checks for every dense token in the list if they are missing from the store or, at the contrary, if their associated number is strictly less than their density expressed in the list. Again the first negative response to this check interrupts the process and leads to a failure of the *nask* function. As for the Dense Bach interpreter, two additional functions complete the code, one for printing the contents of the store, and the second for clearing it. Likewise, an object *bb* of type *VDenseBachStore* provides a *reset* function to also clear the store. The complete code of the implementation of these functions is available in Section D.1.3 of Appendix D.

9.1.4 The simulator

The implementation of the simulator still consists in the programming of a function *run_one*, that performs a transition step with respect to the operational semantics of section 5.1.2. The global strategy of the function for the sequential, the parallel and the choice compositions stay unchanged with respect to the code developed for respectively the BachT language and the Dense Bach language. Only the types of the manipulated expressions have been adapted, now in a form of vectors of dense bach tokens. We refer the reader to Sections 8.1.4 and 8.3.4 for the details of the function, and to Section D.1.4 of Appendix D for a complete listing of its code.

9.1.5 Using the command-line interpreter

The full code of the Vectorized Dense Bach interpreter is listed in section D.1 of appendix D. Using the facilities provided by Scala to write postfix notation, Figure 9.1 presents the result of the computation of the following expression:

$$(get(t(2), s(3)) \ ; \ tell(u(3), a(4)) \ || \ tell(t(3), s(4)) \ ; \ ask(u(2), a(2)))$$

This expression is a parallel composition of two sequentially composed subagents. Starting from an empty store, the first subagent cannot execute its first primitive *get(t(2),s(3))*, but the first primitive *tell(t(3),s(4))* of the second subagent can be executed successfully, putting three instances of t and four of s on the store. As the store does not contain any instances of u, the second primitive *ask(u(2),a(2))* of the second subagent cannot be executed. But the present contents of the store permits now the execution of the primitive *get(t(2),s(3))* of the first subagent. This results in the retrieval of two instances of t and three of s, leaving the store with one instance of t and one of s. The second primitive *tell(u(3),a(4))* of the first subagent

```
Welcome to Scala version 2.11.7 (OpenJDK Server VM, Java 1.7.0_95).
Type in expressions to have them evaluated.
Type :help for more information.

scala> :load dbach-cli.scala
Loading dbach-cli.scala...
...

scala> ag run "(get(t(2),s(3));tell(u(3),a(4))||tell(t(3),s(4));ask(u(2),a(2)))"
{ t(3) s(4) }
{ t(1) s(1) }
{ t(1) s(1) a(4) u(3) }
{ t(1) s(1) a(4) u(3) }
Success
scala>
```

Figure 9.1: Running the Vectorized Dense Bach command line interperter (1)

can also be executed, adding three instances of u and four of a on the store, together to the t and the s already present. Finally with such contents of the store, the primitive *ask(u(2),a(2))* of the second subagent can now be executed successfully, leaving the store unmodified.

As a second example, we propose to evaluate the following expression:

$$(ask(r(2), t(1)) \; ; \; tell(u(3))) \; + \; (nask(s(2), t(4)) \; ; \; get(t(1), a(1))) \; || \; (tell(t(3)) \; ; \; tell(a(3)))$$

In the choice composition, the left member cannot be executed, as there is no token r and t on the store. The right member is a parallel composition, that can start randomly with either *nask(s(2),t(4))* or *tell(t(3))*. The primitive *tell(t(3))* being presently chosen as first step, 3 instances of token t are placed on the store. The two primitives *nask(s(2),t(4))* and *tell(a(3))* constituing a successful first step, in our example the choice is made on *nask(s(2),t(4))*, that leaves the store unchanged. Between the primitive *get(t(1),a(1))* and the primitive *tell(a(3))*, only the second one can be executed successfully, adding three instances of the token a on the store. Finally, the primitive *get(t(1),a(1))* is executed, retrieving one unit of each token t and a. The total result of the execution is available in Figure 9.2.

```
Welcome to Scala version 2.11.7 (OpenJDK Server VM, Java 1.7.0_95).
Type in expressions to have them evaluated.
Type :help for more information.

scala> :load dbach-cli.scala
Loading dbach-cli.scala...
...

scala> ag run "(ask(r(2),t(1));tell(u(3))) +
                    (nask(s(2),t(4));get(t(1),a(1)) || tell(t(3));tell(a(3)))"
{ t(3) }
{ t(3) }
{ t(3) a(3) }
{ t(2) a(2) }
Success
```

Figure 9.2: Running the Dense Bach command line interpreter (2)

9.2 A Command Line Simulator for Vectorized Dense Bach

9.2.1 Introduction

As for the BachT and the Dense Bach languages, the development of a command line simulator is motivated by the necessity to provide a specific environment for the Vectorized Dense Bach language, allowing real parallel executions and a real competition or alternative in a choice. Moreover any agent blocked in its execution could be awakened by the concurrent execution of another agent. As for the other simulators, the design relies essentially on the use of threads. The kind of interface we obtain is as follows:

```
Welcome to Vectorized Dense Bach version 1.
 Type in agents to evaluate them.

VDBach> ask(t(1),u(2)) + get(r(2),u(2)).
VDBach> >> Request 1 launched

VDBach>
```

In this example, starting from an empty store, no subagent in this choice composition can be executed. The request associated with it has been launched and waits for an evolution of the store that can release the blocking. Let us suppose for instance that a second agent *tell(t(3),r(3),u(3))* is now introduced, associated with a second request. It brings enough elements to solve both subagents. Nevertheless, a random choice will be done in favour of one

of them. In this case, the second subagent has been chosen, leading to the following situation where both the first and second requests are solved:

```
 Welcome to Vectorized Dense Bach version 1.
 Type in agents to evaluate them.

VDBach> ask(t(1),u(2)) + get(r(2),u(2)).
VDBach> >> Request 1 launched

VDBach> tell(t(3),r(3),u(3)).
VDBach> >> Request 2 launched

VDBach> >> tell(t(3),r(3),u(3)) successfully terminated
        >> store : { t(3) r(3) u(3) }

VDBach> >> get(r(2),u(2)) successfully terminated
        >> store : { t(3) r(1) u(1) }

VDBach> >> Request 1 successfully terminated

VDBach> >> Request 2 successfully terminated

VDBach>
```

These two examples show the flexibility in the behaviour of the command line simulator, as they also show the capacity of the Vectorized Dense Bach language to handle simultaneoulsy many instances of different dense tokens. As presented in the introduction of Chapter 5, this capacity is clearly an extension of the Dense Bach language. Nevertheless this difference will have a limited impact on the code of the new command line simulator with regard the one developed for the Dense Bach language.

As we did for the interpreter, the structure of the data has been adapted to introduce the notion of dense token that can be recorded in a list. The rest of the global structure still consists of a parser of agents, a class managing the execution of agents, an implementation of the store, and a global object containing the main method. Apart from the necessity to handle the new structures of data, these elements will be slightly modified, as it will be presented in the next sections. The code of the two abstract classes *dtExpr* and *vDBachExpr* is provided in Figure 9.3.

```
class dtExpr
case class dt(tok: String, dens: Int) extends dtExpr

class vDBachExpr
case class vdbach_ast_empty_agent() extends vDBachExpr
case class vdbach_ast_primitive(primitive: String,
                                lDenseToken: List[dt]) extends vDBachExpr
case class vdbach_ast_agent(op: String, agenti: vDBachExpr,
                                  agentii: vDBachExpr) extends vDBachExpr
```

Figure 9.3: The abstract Dense Token class and the abstract Vectorized Dense Bach data class

9.2.2 The parser

The parser used for the Vectorized Dense Bach command line simulator is exactly the same as the one used for the interpreter. Concretely this means that two definitions, one to represent a dense token and the second to represent a list of dense tokens come in addition to the definitions of a token, and of the density. Furthermore, the definitions of the primitives have been adapted to take into account the structure of list of dense tokens. Concerning the operators, their priority order stays unchanged, with the highest one to the sequentiality and the lowest one to the choice. The code is no more presented here but is available in annex (see Section D.2 of Appendix D).

9.2.3 Executing agents

The *VDB_Exec* class is in charge of the execution of the Vectorized Dense Bach agents. As for the other command line simulators, the class receives a parsed agent, that is executed by an *exec* function, according to its nature, i.e. a primitive, a sequentially composed agent, an agent resulting from a parallel composition, or from a choice composition. For the primitive, an *exec_primitive* function processes the parsing result, but with a pair of parameters adapted to the Vector of Dense tokens structure: on the one hand, a primitive type, and on the other hand, a list of dense tokens. Following the type of the primitive, the function *exec_primitive* invokes the classical functions defined for the execution of the *tell*, the *get*, the *ask* or the *nask*, but now adapted to handle a list of dense tokens. These functions are still used for the management of the store, and are presented in Section 9.2.4. Finally, after the primitives, the cases of a parsed agent resulting from a parallel composition, a sequential composition or a choice composition are processed exactly as they are for the previous Dense Bach or BachT command line simulators. In particular, for a parallel composition, threads are associated with both agents *ag_i* and *ag_ii*

acting in the composition, with the obligation to be joined at the end of their execution. For a choice composition, the selection of the executable subagent among those that are part of the choice, is based on the first step. A function *ag_first_steps* associates with every subagent a pair constitued by a first step primitive, followed by a continuation. The continuations being stored in a vector, their associated first steps primitives are stored in an indexed list, with the index pointing to their corresponding continuation in the vector. A random permutation of the second list is provided to a function *exec_l_choice*, that returns the first executable subagent selected on its first step. Again due to the great similarity with the class *DB_Exec*, the complete code of the *VDB_Exec* class with all the previous cited functions is not presented here but is available in annex (see Section D.2 of Appendix D).

9.2.4 The store

The store is also represented as a mapping, associating a token with a number representing the number of its occurrences on the store. As mentioned in section 9.2.3, four functions define the Vectorized Dense Bach primitives *tell*, *get*, *ask* and *nask*, that handle now lists of dense tokens. This implies that the very nature of these primitives is preserved, except that their respective action on the store has to be repeated for every dense token present in the list that they handle. Furthermore in a logic of threads, to perform their action, these function have to take the lock on the store. Among them only those that modify the store – in this case the *tell* and *get* primitives – have to notify their successful action. At the contrary of the *tell* primitive that always succeeds, the three others can face a state of the store that does not permit their complete execution, for some of the dense tokens that belong to their list. In such a case, these primitives have to wait until a modification of the store is notified. To examplify our explanations, Figure 9.4 presents the code of the *get* primitive.

In the continuation of Section 9.2.3 concerning the execution of an agent formed by a choice, a subagent is selected for execution if its first step primitive is checked executable. This is performed by boolean test functions, that check the same conditions as the four previous primitives *tell*, *get*, *ask* and *nask*. When successfully selected they also perform their action on the contents of the store.

Finally the *exec_l_choice* function mentioned in Section 9.2.3 uses the functions *run_l_choice* and *l_choice* defined in the store. The first function *run_l_choice* tests every first step primitive, to determine if it is executable or not. In case of no success, the process waits until a notified modification of the store restarts it. In case of success, the second function *l_choice* returns the index of its continuation in the vector of continuation. These two functions being identical

```
def get(vectDenseTokenList:List[dt]) =  bb.synchronized {
    var list : List[dt] = vectDenseTokenList
    var s : String = reg_list(vectDenseTokenList)
    while(!ag_eval(vectDenseTokenList)) {
        println("Get waiting")
print("VDBach> ")
bb.wait()}
    while(!(list.isEmpty)) {
mapTok(list.head.tok) = mapTok(list.head.tok) - list.head.dens
list = list.tail
    }
    println(">> get("+s+") successfully terminated")
    print("        >> store :")
    print_store
    println()
    print("VDBach> ")
    bb.notifyAll()
    true
    }
```

Figure 9.4: The Vectorized Dense Bach *get* primitive

to their corresponding version for the BachT and Dense Bach command line simulators, their code is not presented here but is available in annex (see Section D.2 of Appendix D), with the complete code of the Vectorized Dense Bach command line simulator.

9.2.5 The main object

The code of the command line simulator finishes with the definition of a main object called *MYSimInLine*. It contains the main function that executes a Vectorized Dense Bach agent. The structure of this function is organized identically to the equivalent functions of the BachT and Dense Bach command line simulators. Based on the content of the command line, a main loop proposes different reactions. The commands *history*, *print* and *clear* are respectively responsible for listing the last five commands, printing the contents of the store, and making an empty store. Any other line that finishes with a dot is considered as an agent to be parsed and executed, with a thread associated to it. Every launched thread receives a number that permits to follow its complete execution. Being similar to what we did previously the code of this *main* function is not reproduced in the main text of this thesis but is listed in annex (see Section D.2 of Appendix D).

```
 Welcome to Vectorized Dense Bach version 1.
 Type in agents to evaluate them.

VDBach> tell(t(4),r(2));get(t(2),r(1)) || nask(s(2),t(1));ask(t(2)).
VDBach> >> Request 1 launched

VDBach> >> tell(t(4),r(2)) successfully terminated
        >> store : { t(4) r(2) }

VDBach> >> get(t(2),r(1)) successfully terminated
        >> store : { t(2) r(1) }

VDBach>
```

Figure 9.5: The execution of `tell(t(4),r(2));get(t(2),r(1))`

9.2.6 Using the Vectorized Dense Bach command line simulator

In addition to the example of execution provided in the introduction, let us now suppose that we want the command line simulator to evaluate the following expression starting from an empty store:

$$tell(t(4), r(2)) \ ; \ get(t(2), r(1)) \ || \ nask(s(2), t(1)) \ ; \ ask(t(2))$$

In this parallel composition, the first steps of both subagents are both executable. The execution starts with the first subagent, that produces a store containing two instances of token t and one of token r. The two tokens t block the execution of the *nask(s(2),t(1))* primitive, as shown in Figure 9.5.

A second request for introducing a new agent *get(t(2))* to retrieve the two tokens t permits the execution of the *nask(s(2),t(1))* primitive. Nevertheless, the absence of tokens t on the store suspends the execution of the *ask(t(2))* primitive. Figure 9.6 shows this second step of execution.

A third request introducing three instances of the token t permits to execute the *ask(t(2))* primitive, and closes the execution of the parallel agent, as being the first request introduced. Figure 9.7 shows the last result of the execution.

```
 Welcome to Vectorized Dense Bach version 1.
 Type in agents to evaluate them.

VDBach> tell(t(4),r(2));get(t(2),r(1)) || nask(s(2),t(1));ask(t(2)).
VDBach> >> Request 1 launched

VDBach> >> tell(t(4),r(2)) successfully terminated
        >> store : { t(4) r(2) }

VDBach> >> get(t(2),r(1)) successfully terminated
        >> store : { t(2) r(1) }

VDBach> get(t(2)).
VDBach> >> Request 2 launched

VDBach> >> get(t(2)) successfully terminated
        >> store : { t(0) r(1) }

VDBach> >> nask(s(2),t(1)) successfully terminated
        >> store : { t(0) r(1) }

VDBach>
```

Figure 9.6: The execution of `nask(s(2),t(1))`

```
 Welcome to Vectorized Dense Bach version 1.
 Type in agents to evaluate them.

VDBach> tell(t(4),r(2));get(t(2),r(1)) || nask(s(2),t(1));ask(t(2)).
VDBach> >> Request 1 launched

VDBach> >> tell(t(4),r(2)) successfully terminated
        >> store : { t(4) r(2) }

VDBach> >> get(t(2),r(1)) successfully terminated
        >> store : { t(2) r(1) }

VDBach> get(t(2)).
VDBach> >> Request 2 launched

VDBach> >> get(t(2)) successfully terminated
        >> store : { t(0) r(1) }

VDBach> >> nask(s(2),t(1)) successfully terminated
        >> store : { t(0) r(1) }

VDBach> >> Request 2 successfully terminated

VDBach> tell(t(3)).
VDBach> >> Request 3 launched

VDBach> >> tell(t(3)) successfully terminated
        >> store : { t(3) r(1) }

VDBach> >> ask(t(2)) successfully terminated
        >> store : { t(3) r(1) }

VDBach> >> Request 1 successfully terminated

VDBach> >> Request 3 successfully terminated

VDBach>
```

Figure 9.7: The execution of `ask(t(2))`

9.3 A command-line interpreter for MRT

9.3.1 Introduction

As for the Vectorized Dense Bach language we develop in the following section a command-line interpreter for the MRT language. As explained in Section 3.1, this language consists in re-writing pre-condition multi-sets in post-condition multi-sets. As for the previous interpreter, the code is organised in four files : the data structure, the parser, the store and the simulator. Nevertheless some of them need to be adapted to take into account the structure of muti-sets.

For MRT, the abstract data file defines two classes. The first one extends the notion of expression, with two classes mr_tp and mr_tn that characterize a token preceeded by a plus sign or a minus sign. The second class is dedicated to the abstract syntax tree as for the other language interpreters. The definitions of the two abstract classes are as follows:

```
class mrExpr
case class mr_tp(tok: String) extends mrExpr
case class mr_tn(tok: String) extends mrExpr

class mrAgExpr
case class mrt_ast_empty_agent() extends mrAgExpr
case class mrt_ast_primitive(Pre: List[mrExpr],
                             Post: List[mrExpr]) extends mrAgExpr
case class mrt_ast_agent(op: String, agenti: mrAgExpr,
                             agentii: mrAgExpr) extends mrAgExpr
```

9.3.2 The parser

The parser defines the class *MRTParsers* that extends the class *RegexParsers*. Its construction is adapted to take into account the fact that tokens are annotated and that the language contains only a primitive written as *Preconditions* → *Postconditions*, with the pre- and post-conditions being lists of annotated tokens. The definition of tokens being the same as the one for the BachT language, an annotated token is a sequential composition of a sign followed by a token, defined as follows:

```
def atoken : Parser[mrExpr] = "+" ~ token ^^ {
                          case _ ~ vtoken => mr_tp(vtoken)} |
                              "-" ~ token ^^ {
                          case _ ~ vtoken => mr_tn(vtoken)}
```

Using this definition, we can then define *atokenList* as a non-empty list of annotated tokens in the expected way with the Scala `rep` operator :

```
def atokenList: Parser[List[mrExpr]] = atoken ~ rep("," ~ atokenList) ^^ {
                  case vatoken ~ List() => List(vatoken)
                  case vatoken ~ List(op~lvat) => List(vatoken):::lvat
                  }
```

Equipped with this definition, we can then specify a pre-condition or a post-condition as formed from the set bracket being opened and immediately closed, which denotes an empty list of annotated tokens, or as containing inside the brackets at least one annotated token, and hence a *atokenList* :

```
def preMR: Parser[List[mrExpr]] =
     "{" ~ "}" ^^ { case _ ~_   => List() } |
     "{" ~ atokenList ~ "}" ^^ { case _~lvat~_ => lvat }

def postMR: Parser[List[mrExpr]] =
     "{" ~ "}" ^^ { case _ ~ _  => List() } |
     "{" ~ atokenList ~ "}" ^^ { case _~lvat~_ => lvat }
```

Reading a primitive then consists in successively reading the opening parenthesis `(`, then a pre-condition `preMR`, then the arrow `->` then a post-condition `postMR` and finally the closing parenthesis `)`. The code of the primitive is as follows:

```
def primitive : Parser[mrAgExpr] = "(" ~ preMR ~ "->" ~ postMR ~ ")" ^^ {
                          case _ ~ vatokenList1 ~ _ ~ vatokenList2 ~ _ =>
                                    mrt_ast_primitive(vatokenList1,vatokenList2)
                                                }
```

These definitions are completed by the same code as for the other languages, to define composed agents with the same priorities given to the sequential, parallel and non-deterministic choice operators. The complete code of the parser is not presented here but is available in Section E.1.2 of Appendix E.

9.3.3 The store

Given the specific syntax of the MRT primitive, handling the store requires new treatments. The most notable one is that, as the pre- and post-conditions may contain several occurrences of tokens, mappings are introduced to associate each token with the required number of multiplicity, both for the pre- and post-conditions. More precisely, four mappings are used :

- `thePosPre` for the tokens appearing positively in the pre-condition
- `theNegPre` for the tokens appearing negatively in the pre-condition
- `thePosPost` for the tokens appearing positively in the post-condition
- `theNegPost` for the tokens appearing negatively in the post-condition

These maps are constructed by means of two auxiliary methods `add_to_pre_lists` and `add_to_post_lists`, which respectively add the annotated atoms of the pre- and post-conditions to the mappings associated with them. The code for these mappings and methods is presented in Figures 9.8 and 9.9.

Based on these mappings, and depending of the sign and the nature (pre or post) of the conditons, differents methods, recalling the Dense Bach primitives, are called. For the negative pre-conditions, a method *nask* is invoked to verify the absence of any of the concerned token. For the positive pre-conditions, a method *ask* is invoked, to check for the effective presence of at least the registered multiplicity of the concerned token on the store. For the negative post conditions, a method *get* is invoked for retrieving from the store the registered multiplicity of the concerned token. Finally, for the positive post conditions, a method *tell* is invoked to place the required number of the concerned token on the store. The code for the primitives is presented in Figure 9.10.

With the help of these methods, the evaluation of a pre-condition consists in constructing the two mappings associated with it and in asking the tokens positively marked with the required number together with negatively asking for the negatively marked tokens with respect to the associated number. Similarly, the evaluation of a post-conditions consists in constructing the two mappings associated with it and in respectively telling and getting the positively and negatively marked tokens. The code for these two evaluations is listed in Figure 9.11. Note that for optimization purposes, we continue the evaluation of a pre- or post-conditions until it becomes false. This is materialized through the variables `pre_eval` and `post_eval`. These variables are also returned as results of the evaluations.

The execution of a MRT primitive is then quite simple. It consists in evaluating the pre-condition and in case of success of evaluating the post-condition. Note that since the evaluation of the pre-condition does not modify the store, no action needs to be undertaken in case of failure. The code is given in Figure 9.11.

As for the other languages, two auxiliary methods are provided to print the contents of the store and for clearing it. A companion object is also given to the resulting `MrtStore`, named `bb`.

```
var thePosPre = Map[String,Int]()
var theNegPre = Map[String,Int]()

var thePosPost = Map[String,Int]()
var theNegPost = Map[String,Int]()

def add_to_pre_lists(atoken: mrExpr) {

   atoken match {

      case  mr_tp(x) => {
             if (thePosPre.contains(x))
                   { thePosPre(x) = thePosPre(x) + 1 }
             else
                   { thePosPre = thePosPre ++ Map(x -> 1) }
       }

      case  mr_tn(x) => {
             if (! theNegPre.contains(x))
                   { theNegPre = theNegPre ++ Map(x -> 1) }
       }

      case _ => { println("error_in_precondition") }

    }
}
```

Figure 9.8: Mappings associated with pre- and post-conditions

The complete code of the store class *MrtStore* is available in Section E.1.3 of Appendix E.

9.3.4 The simulator

For MRT, as for the other languages, the simulator is articulated around the *run_one* method which performs a transition step. On that basis, the *mr_exec_all* repeatedly executes one step until the empty agent is reached or failure is produced.

Being similar, the code is not reproduced inside the main text of this thesis. It is however listed in Section E.1.4 of Appendix E.

9.3.5 Using the command-line interpreter

The complete code of the command-line interpreter is listed in Section E.1.4 of Appendix E. As for the other languages, it is organized in four files, three for the parser, the store and the simulator, together with one for the definition of the case classes. For the ease of use, they have been concatenated to form a single file, called *complete_simulator.scala*.

```
def add_to_post_lists(atoken: mrExpr) {

   atoken match {

      case  mr_tp(x) => {
             if (thePosPost.contains(x))
                   { thePosPost(x) = thePosPost(x) + 1 }
             else
                   { thePosPost = thePosPost ++ Map(x -> 1) }
       }

      case  mr_tn(x) => {
             if (theNegPost.contains(x))
                   { theNegPost(x) = theNegPost(x) + 1 }
             else
                   { theNegPost = theNegPost ++ Map(x -> 1) }
       }

      case _ => { println("error_in_postcondition") }

    }
}

def pre_to_prelist(latoken: List[mrExpr]) {

     thePosPre = Map[String,Int]()
     theNegPre = Map[String,Int]()

     for (a <- latoken) { add_to_pre_lists(a) }

}

def pos_to_postlist(latoken: List[mrExpr]) {

     thePosPost = Map[String,Int]()
     theNegPost = Map[String,Int]()

     for (a <- latoken) { add_to_post_lists(a) }

}
```

Figure 9.9: Mappings associated with pre- and post-conditions (continued)

```
 def ask(token:String,number:Int): Boolean = {

    if (theStore.contains(token))
        if (theStore(token) >= number) { true }
        else { false }
    else { false }
 }

 def nask(token:String,number:Int):Boolean = {

    if (theStore.contains(token))
           if (theStore(token) >= number) { false }
           else { true }
    else { true }
 }

 def tell(token:String,number:Int):Boolean = {

    if (theStore.contains(token))
      { theStore(token) = theStore(token) + number }
    else
      { theStore = theStore ++ Map(token -> number) }
    true
 }

def get(token:String,number:Int):Boolean = {

    if (theStore.contains(token))
           if (theStore(token) >= number )
              { theStore(token) = theStore(token) - number
                true
              }
           else { println("density not enough")
                  false }
    else {  println("no token present")
            false }
 }
```

Figure 9.10: Elementary primitives for the pre- and post-conditions

```
def Pre(latoken: List[mrExpr]):Boolean = {

     var pre_eval = true

     pre_to_prelist(latoken)

     for ( (t,v) <- thePosPre ) {
        if ( pre_eval ) { pre_eval = ask(t,v) }
     }

     for ( (t,v) <- theNegPre ) {
        if ( pre_eval ) { pre_eval = nask(t) }
     }

     pre_eval

}

def Post(latoken: List[mrExpr]):Boolean = {

     var post_eval = true

     pos_to_postlist(latoken)

     for ( (t,v) <- thePosPost ) {
        if ( post_eval ) { post_eval = tell(t,v) }
     }

     for ( (t,v) <- theNegPost ) {
        if ( post_eval ) { post_eval = get(t,v) }
     }

     pos_eval
}

def execution_primitive(mrPre: List[mrExpr],mrPost: List[mrExpr]):Boolean = {
     if(Pre(mrPre)) { Post(mrPost)
                      true }
     else { false }
}
```

Figure 9.11: Evaluation of the pre- and post-conditions

```
dda$scala
Welcome to Scala version 2.11.7 (OpenJDK Server VM, Java 1.7.0_131).
Type in expressions to have them evaluated.
Type :help for more information.

scala> :load complete_simulator.scala
Loading complete_simulator.scala...
...

scala>ag run "({+s} -> {+t}) + ({-s} -> {+u,+u})"
{ u(2) }
scala>
```

Figure 9.12: Running the MRT command-line interperter

Using the postfix notation that was already used for the other languages, Figure 9.12 illustrates the computation of

$$(\{+s\} \rightarrow \{+t\}) \ + \ (\{-s\} \rightarrow \{+u, +u\})$$

As it requires in its pre-condition the presence of token s and as the store is initially empty the first alternative cannot be executed. In contrast, the second alternative specifies the absence of s in its pre-condition and can thus be executed. This leads to telling twice token u, as stated in the post-condition.

As a second example, let us consider the following agent :

$$((\{+r\} \rightarrow \{+t\}); (\{+t\} \rightarrow \{+u\})) \ \| \ ((\{-s\} \rightarrow \{+r\}); (\{+u\} \rightarrow \{-r, -t\}))$$

As s is absent from the store, initially empty, its computation consists of first executing the $(\{-s\} \rightarrow \{+r\})$ transition, which produces r on the store, then the transition $(\{+r\} \rightarrow \{+t\})$ which produces t on the store, then the transition $(\{+t\} \rightarrow \{+u\}$ which produces u on the store and finally the transition $(\{+u\} \rightarrow \{-r, -t\})$ which removes r and t from the store. This is indeed what is delivered by the command-line interperter, as depicted in Figure 9.13.

The third example is related to the chemical world, namely by the representation of the reaction producing water with hydrogen and oxygen. If a token *h2* represents a molecule of hydrogen (H_2), a token *o2* a molecule of oxygen (O_2) and a token *h2o* a molecule of water (H_2O), then the following processes represents the chemical reaction producing two molecules of water, by consuming two molecules of hydrogen and one of oxygen.

```
dda$scala
Welcome to Scala version 2.11.7 (OpenJDK Server VM, Java 1.7.0_131).
Type in expressions to have them evaluated.
Type :help for more information.

scala> :load complete_simulator.scala
Loading complete_simulator.scala...
...

scala>ag run "(({+r}->{+t});({+t}->{+u})) || (({-s}->{+r});({+u}->{-r,-t}))"
{ r(1) }
{ r(1) t(1) }
{ r(1) t(1) u(1) }
{ r(0) t(0) u(1) }
Success
scala>
```

Figure 9.13: The MRT interperter on a parallel agent

$$\{+h2, +h2, +o2\} \rightarrow \{-h2, -h2, -o2, +h2o, +h2o\}$$

As easily observed the pre-conditions require two times a token h2 and one time a token o2. The post-conditions destroy those molecules and produce two times a token h2o, representing the two molecules of water.

9.4 A command line simulator for MRT

9.4.1 Introduction

We propose now to develop a command line simulator for the MRT language, that offers the user a specific environment, with real parallel execution or real competition of alternative in a choice. The global structure of the code is still composed of a parser, an executable class, a representation of the store, and finally an object that performs the execution of a MRT agent. The structure of the data is identical to the one used for the MRT interpreter. It is composed of two abstract classes *mrAgExpr* and *mrExpr*. The case classes of the first one represent the structure of a parsed MRT agent, and the case classes of the second one represent positively

annotated token, and negatively annotated tokens. The code of these abstract classes are available in annex (see Section E.2 of Appendix E).

9.4.2 The parser

The parser is identical to the one used for the interpreter. It specifies annotated tokens, that are grouped in lists, that themselves are used to form pre-condition and post-condition lists. The reading of a primitive can then be expressed as reading the opening parenthesis `(`, then a pre-condition `preMR`, then the arrow `->` then a post-condition `postMR` and finally the closing parenthesis `)`. The definitions of composed agents stay unchanged, with the same priorities given to the sequential, parallel and non-deterministic choice operators. The complete code of the parser is available in Section E.2 of Appendix E.

9.4.3 Executing agents

The *MRT_Exec* class is in charge of the execution of the MRT agents. The class receives a parsed agent *current_agent*, that is executed by an *exec* function, according to its nature, i.e. a primitive, a sequentially composed agent, an agent resulting from a parallel composition, or from a choice composition. For the primitive, an *exec_primitive* function processes the parsing result, but with four parameters in the form of mappings, that represent the lists of the positive and negative tokens for the pre-conditions and those for the post-conditions. A function *eval_pre* defined on the store evaluates the pre-conditions of the primitive. In case of success, all the post-conditions are executed. In case of failure, the execution of the primitive is suspended, waiting for a notification of a modification of the store. Four functions that recall the Dense Bach primitives *tell*, *get*, *ask* and *nask* are used in the evaluation of the pre-conditions, and the execution of the post-conditions. They are defined on the store and presented in Section 9.4.4. The code of the function *exec_primitive* is as follows:

```
def exec_primitive(mrPrePos: Map[String,Int], mrPreNeg: Map[String,Int],
     mrPostPos: Map[String,Int], mrPostNeg: Map[String,Int]) = bb.synchronized {

          var post_eval = true
          while(!(bb.eval_pre(mrPrePos,mrPreNeg))) {bb.wait()}
          for ( (t,v) <- mrPostPos ) {
              if ( post_eval ) { post_eval = bb.tell(t,v) }
          }
          for ( (t,v) <- mrPostNeg ) {
              if ( post_eval ) { post_eval = bb.get(t,v) }
```

```
        }
        bb.notifyAll()
        true
}
```

The processing of parsed agents resulting from a parallel composition, a sequential composition or a choice composition is done as in the previous command line simulators. In particular, for a parallel composition, threads are associated with both agents *ag_i* and *ag_ii* acting in the composition, with the obligation to be joined at the end of their execution. For a choice composition, the selection of the executable subagent among those that are part of the choice, is based on the first step. Two lists are constructed, the first one grouping the first steps primitives, and the second one grouping their continuations. A random permutation of the first list is given to a function *exec_l_choice* that returns the continuation of the first executable subagent selected on its first step. Again the great similarity of the *MRT_Exec* class with its corresponding class of the *VDB_Exec* class makes useless to list the code. It is available in annex (see Section E.2 of Appendix E).

9.4.4 The store

The store is again represented as a mapping, associating a token with a number representing the number of its occurrences on the store. As mentioned in Section 9.4.3, four functions similar to the Dense Bach primitives *tell*, *get*, *ask* and *nask* are defined. All of them have two parameters: a string for the token name, and an integer for its number of occurrences. These functions are used for the evaluation of the pre-conditions or the execution of the post-conditions. For the negative pre-conditions, the *nask(token,number)* is invoked to verify that *number* is strictly smaller than the registered multiplicity of the concerned token on the store. For the positive pre-conditions, a method *ask(token,number)* is invoked, to check that *number* has at least the same value as the registered multiplicity of the concerned token on the store. For the negative post conditions, a method *get(token,number)* is invoked for retrieving *number* from the registered multiplicity of the concerned token on the store. Finally, for the positive post conditions, a method *tell(token,number)* is invoked to place the required *number* of the concerned token on the store.

Finally the store defines the function *l_choice* used by the function *exec_l_choice* invoked in Section 9.2.3. The function returns the continuation of an executable first step. Again due to the great similarity between this code and the equivalent code of the Vectorized Dense Bach

command line simulator, we do not list it in the main text of the thesis. However it is available in annex (see Section E.2 of Appendix E).

9.4.5 The main object

The code of the command line simulator finishes with the definition of a main object called *MYSimInLine*. This code is identical to the corresponding code developed for the Vectorized Dense Bach command line simulator. This means that it provides also the same commands *history*, *print* and *clear*, that are respectively responsible for listing the last five commands, printing the contents of the store, and making an empty store. Any other command is considered as an agent, which is then submitted to the parser, and executed with an associated thread. The code of the *MYSimInLine* object is available in annex (see Section E.2 of Appendix E).

9.4.6 Using the MRT command line simulator

We now describe some examples of the use of the MRT command line simulator. Let us suppose we want to execute the following agent, starting from an empty store:

```
    ({+t,+t,+s,+s,+s}->{-t,-t,-s,-s,-s});({}->{+u,+u,+u,+a,+a})
||  ({}->{+t,+t,+t,+s,+s,+s,+s});({+u,+u,+a,+a}->{}).
```

This agent consists in a parallel composition of two subagents, each of them being a sequential composition. Let us explore its execution. In the first step primitive of the first subagent, the retrieval of two tokens t and three tokens s cannot be executed, as the pre-conditions are not fulfilled. In the second subagent, the first step is executable, as it consists in depositing three instances of t and four of s. With such a content of the store, the first step primitive of the first subagent becomes now executable, with a retrieval of two tokens t and three tokens s. This primitive is sequentially followed by a primitive that puts three instances of t and four of s on the store. At this stage, the left part of the parallel composition is now completely executed. The last primitive of the second subagent consisting in verifying the presence of two tokens u and of two tokens a on the store is executable, its execution ends successfully the execution of the parallel agent. The final result is a store with one token t, one token s, two tokens a and three tokens u. The answer of the command line simulator is as follows:

```
 Welcome to MRT version 1.
 Type in agents to evaluate them.

MRT> ({+t,+t,+s,+s,+s}->{-t,-t,-s,-s,-s});({}->{+u,+u,+u,+a,+a})
   |   || ({}->{+t,+t,+t,+s,+s,+s,+s});({+u,+u,+a,+a}->{}).
MRT> >> Request 1 launched

MRT> >> Request 1 successfully terminated

MRT>   print.
 { t(1) s(1) a(2) u(3) }

MRT>
```

Let us now suppose that we want to execute a choice composed agent of two sequentially composed subagents, starting again from an empty store:

```
({}->{+t,+t});({-s,-s,-t}->{}) + ({+u,+u,+r}->{});({+t,+t,+r}->{-t,-t,-r}).
```

In this choice composition, only the first step of the first subagent can be successfully executed. It provides two tokens t on the store, as follows:

```
 Welcome to MRT version 1.
 Type in agents to evaluate them.

MRT> ({}->{+t,+t});({-s,-s,-t}->{}) + ({+u,+u,+r}->{});({+t,+t,+r}->{-t,-t,-r}).
MRT> >> Request 1 launched

MRT> print.
 { t(2) }

MRT>
```

The second primitive in the first subagent cannot be executed, as it requires no token t on the store. The first request is placed in a waiting state, until a new agent retrieves the two tokens t of the store. This is done by introducing a second request, that permits the complete successful execution of the first one, as follows:

```
 Welcome to MRT version 1.
 Type in agents to evaluate them.

MRT> ({}->{+t,+t});({-s,-s,-t}->{}) + ({+u,+u,+r}->{});({+t,+t,+r}->{-t,-t,-r}).
MRT> >> Request 1 launched

MRT> print.
 { t(2) }

MRT> ({+t,+t}->{-t,-t}).
MRT> >> Request 2 launched

MRT> >> Request 2 successfully terminated

MRT>   >> Request 1 successfully terminated

MRT>
```

9.5 Conclusion

As in Chapter 8, we have developed an interpreter and a command line simulator for the Vectorized Dense Bach and the MRT languages with again the objective of providing the reader with interpreters and command line simulators that permit to experience with these two languages. The implementation is very similar to those used for the interpreters and command line simulators developed for BachT and Dense Bach. In particular the design of our implementation is still inspired by Pistache [MdM11], with regards to the case classes, and the use of random generated numbers to handle the parallel and choice compositions. We continue to use threads, with a simpler design based on a global lock on the store.

Chapter 10

Simulations

The interpreters presented in chapters 8 and 9 are useful to experiment with the dense languages. However they offer quite a limited interaction. Indeed, because of their command line nature, they only allow to introduce one agent at a time. The commmand line simulators provide more flexibility and have a more user friendly interface. Nevertheless the modification of the store is only visible through the command line and the user has no control on which alternative to choose.

This chapter presents an alternative way of experimenting with the languages. It is based on graphical elements to illustrate the contents of the store as well as to execute agents, either in an interactive fashion or automatically. As the two previous chapters have evidenced the similarites of the languages and the implementation adaptations to be made, we shall focus in this chapter on Dense Bach only.

10.1 A graphical simulator

10.1.1 Design

10.1.1.1 The store

The central element to animate is without any doubt the store. It indeed consitutes the core of the graphical simulator and the corresponding window is, in fact, the one created when the application is launched. Figure 10.1 shows it at that moment. As can be seen, the central part is composed of the contents of the store, empty at start time but subsequently modified either by the execution of agents or more directly through buttons that allow to add (by telling) or remove (by getting) tokens with a specified density. A button also allows to clear the contents

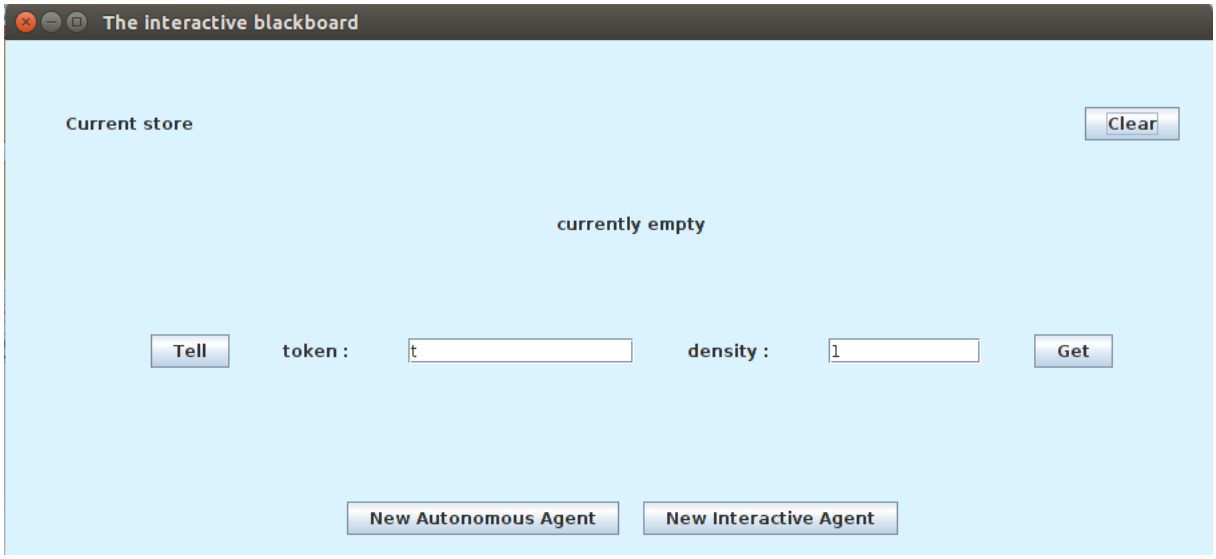


Figure 10.1: The `store` window

of the store by removing all the elements.

The store window also allows to launch two kinds of agents running in parallel, the so-called autonomous and interactive agents. They are described in the next subsections.

10.1.1.2 The interactive agent

When the button "New interactive agent "is pressed in the store window, a new window appears in the form depicted in Figure 10.2. It offers a field for editing the agent to be processed. The button "Submit" parses the introduced agent which is translated, in the "Current agent" part, in an interactive form, with a button being presented for each executable primitive. The user then selects the one he wishes to execute, which has the double effect, on the one hand, of modifying the store and the store window accordingly, and, on the other hand, of updating the "Current agent" part with the new agent to be executed.

As the store can be updated by the execution of concurrent agents or directly by interactions in the store window, a "Refresh" button is offered to update the current agent part with the primitives that can be executed.

10.1.1.3 The autonomous agent

In a similar way, pressing the button "New autonomous agent" results in launching a new window of the type displayed in Figure 10.3. A field is there offered to introduce an agent

Figure 10.2: The `interactive agent` window

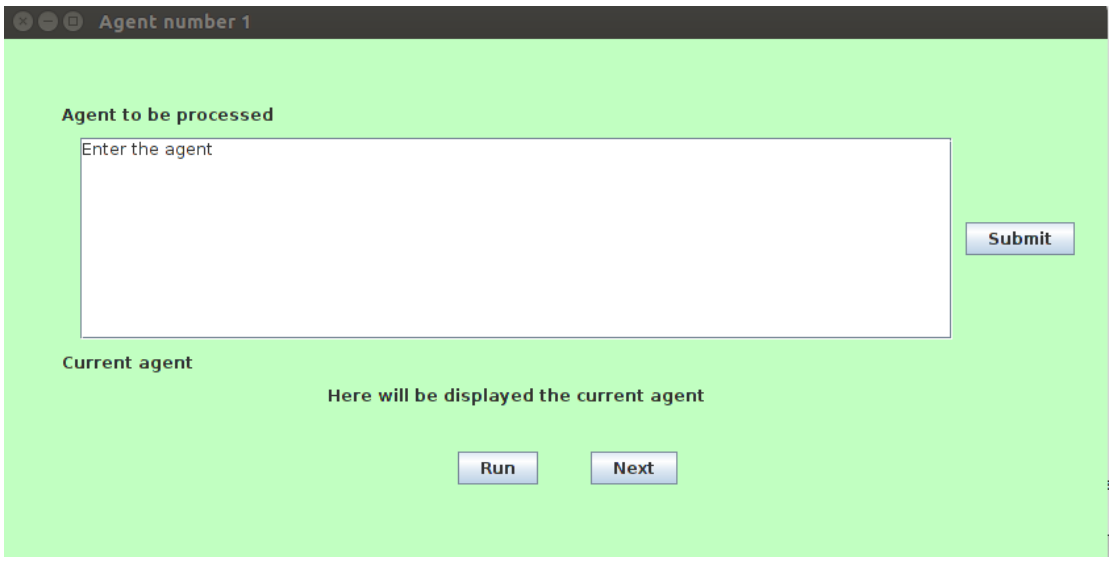


Figure 10.3: The `autonomous agent` window

to be executed. The button "Submit" results in parsing this agent and in displaying it in the "Current agent" part. Two buttons then allow to execute it either in a run or step by step. The "Next" button offers this latter possibility. In case an execution step is possible, the "Current agent" part is updated with the agent resulting from a step and, of course, the store windows is updated accordingly. The agent (and the store) remains the same in case no further computation is possible. Pressing the "Run" button has similar effects but all the steps are made one after the other directly.

It is worth noting that a choice is made randomly by the simulator in case it has to be performed as a result of an agent composed by the non deterministic choice operator or by the parallel composition operator.

10.1.2 Usage

Let us now illustrate the use of the graphical simulator. Suppose we are interested in the following agent :

$$(tell(t(2)) \ + \ tell(v(2))) \ ; (nask(u(3)) \ || \ tell(a(2)))$$

The next two subsections present its evaluation according to the two types of execution : interactive or autonomous.

10.1.2.1 The interactive agent

To take a step by step control of the execution of the proposed agent, the user opens an `interactive agent` window and fills it with the agent of interest, as shown in Figure 10.4.

Figure 10.5 shows the result of pressing the "Submit" button and illustrates the fact that two primitives are executable : `tell(t(2))` and `tell(v(2))`. They correspond to the two choices offered by the non deterministic choice. Note that the primitives in the parallel composition are not yet transformed in buttons, as the choice needs first be evaluated.

Assume the user chooses the first button, representing the execution of the `tell(t(2))` primitive. The store is then adapted as shown in Figure 10.6 and the `interactive agent` window becomes as shown in Figure 10.7. In this figure, the choice of the agent `tell(v(2))` has disappeared and the agent is now reduced to the parallel composition of the two primitives `nask(u(3))` and `tell(a(2))`. Both primitives are now transformed in active buttons, as they can now be evaluated. As a result, through the buttons the user can choose between them.

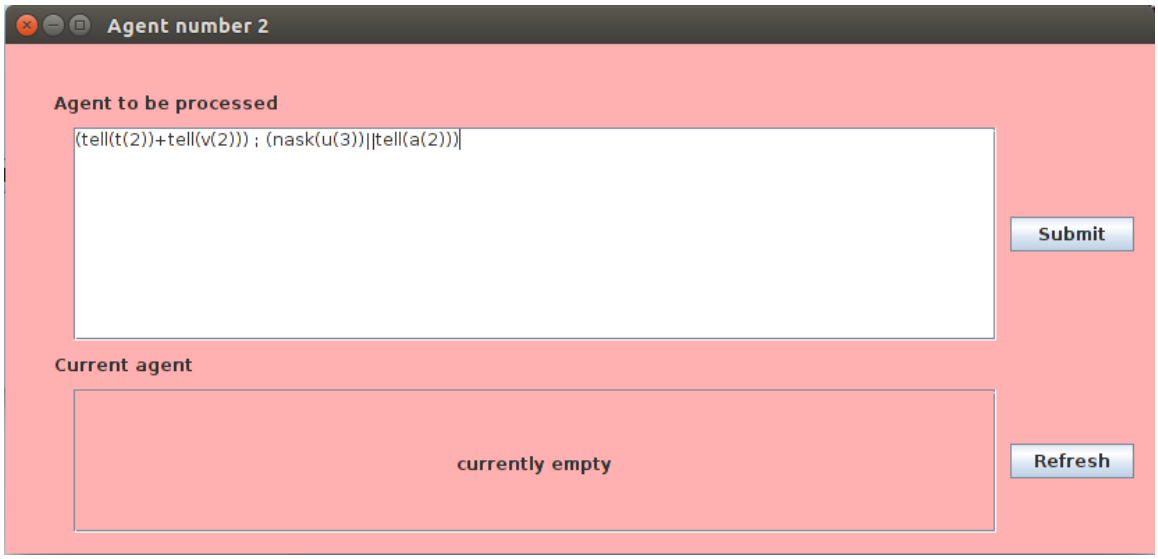


Figure 10.4: The `interactive agent` window for a specific agent

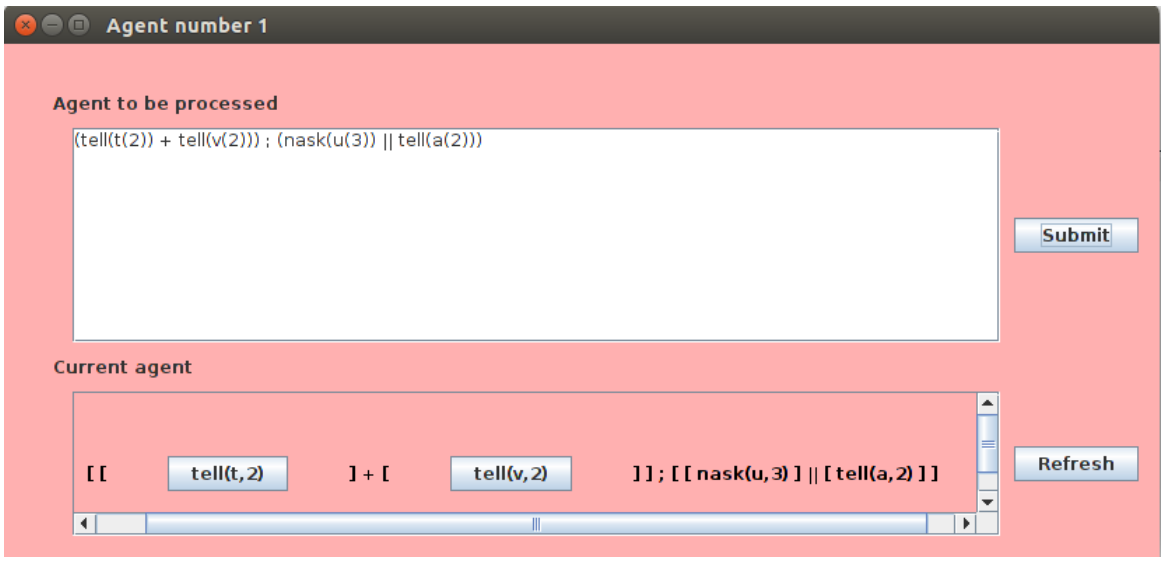


Figure 10.5: The `interactive agent` window for a specific agent

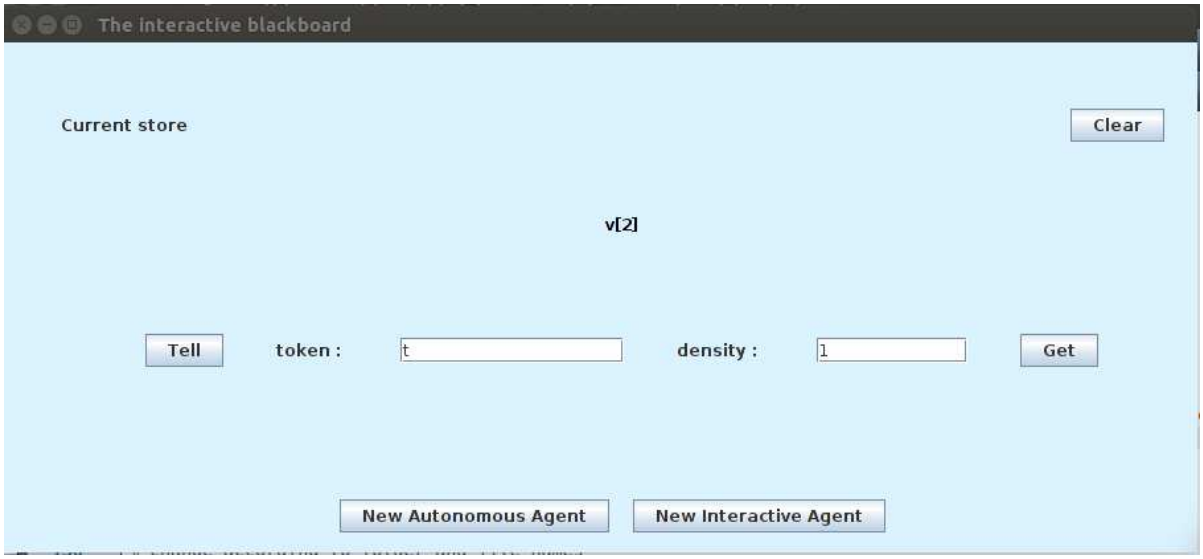


Figure 10.6: The `store` window after the choice of the `tell(v(2))` button in the interactive window

Figure 10.7: The second step of the `interactive agent` window with the remaining parallel composition

Figure 10.8: The third step of the `interactive agent` window after the execution of the `nask(u(3))` primitive.

Assume the user chooses the `nask(u(3))` primitive (which can be executed as there are no token `u` on the store). The `interactive agent` window evolves to the state of figure 10.8 and the store remains unchanged with respect to figure 10.6.

As a final step, the user executes the remaining `tell(a(2))` primitive. Its effect is to modify the store by introducing a token `a(2)` on it. Figure 10.10 represents the new state of the store, and figure 10.9 indicates that the original agent has now been completely executed, which is materialized by the fact that the current agent has become the empty agent.

10.1.2.2 The autonomous agent

The execution with respect to the autonomous form proceeds similarly. The agent is first edited in the text field of the "Autonomous agent" window and then parsed by pressing the "Submit" button. The resulting window is depicted in figure 10.11. Pressing the button "Run" then executes the agent completely, which has the effect of, for instance, executing the `tell(v(2))` and `tell(a(2))` primitives. The resulting store is displayed in the store window of figure 10.12.

As the agent can be executed again, the second execution adds its results to the situation obtained after the first run. The choice between `tell(t(2))` and `tell(v(2))` can produce as a result two other tokens `t` or two tokens `v`. Figures 10.13 and 10.14 respectively represent the induced contents of the store windows. Note that, as explained before, this choice is done randomly by the graphical simulator, without any control of the user.

Figure 10.9: The fourth step of the `interactive agent` window

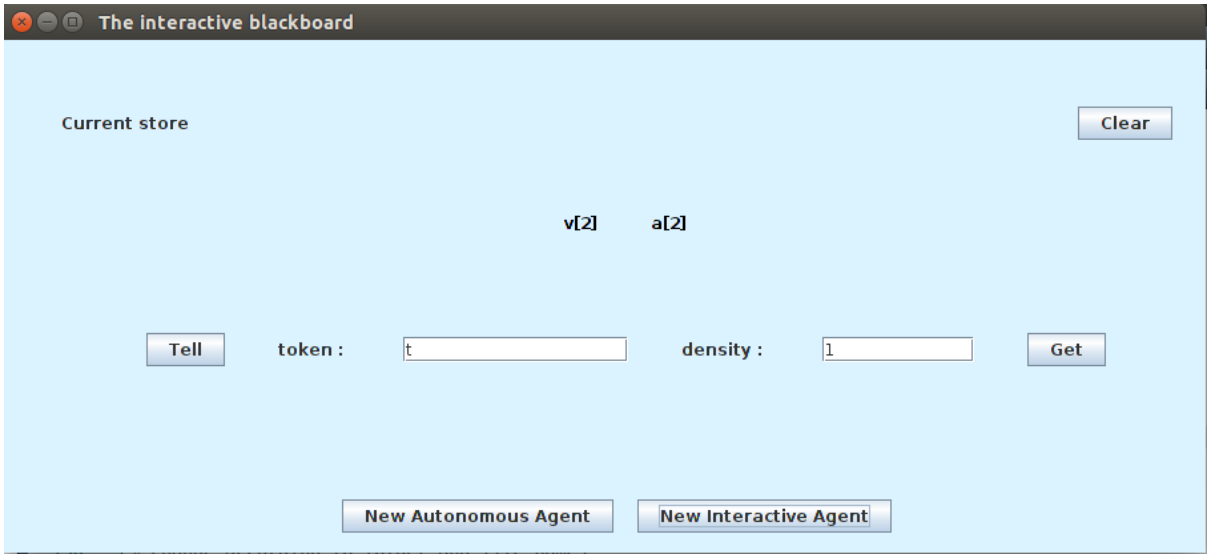


Figure 10.10: The `store` window after the third step of execution of the interactive window

Figure 10.11: The `autonomous agent` window for a specific agent

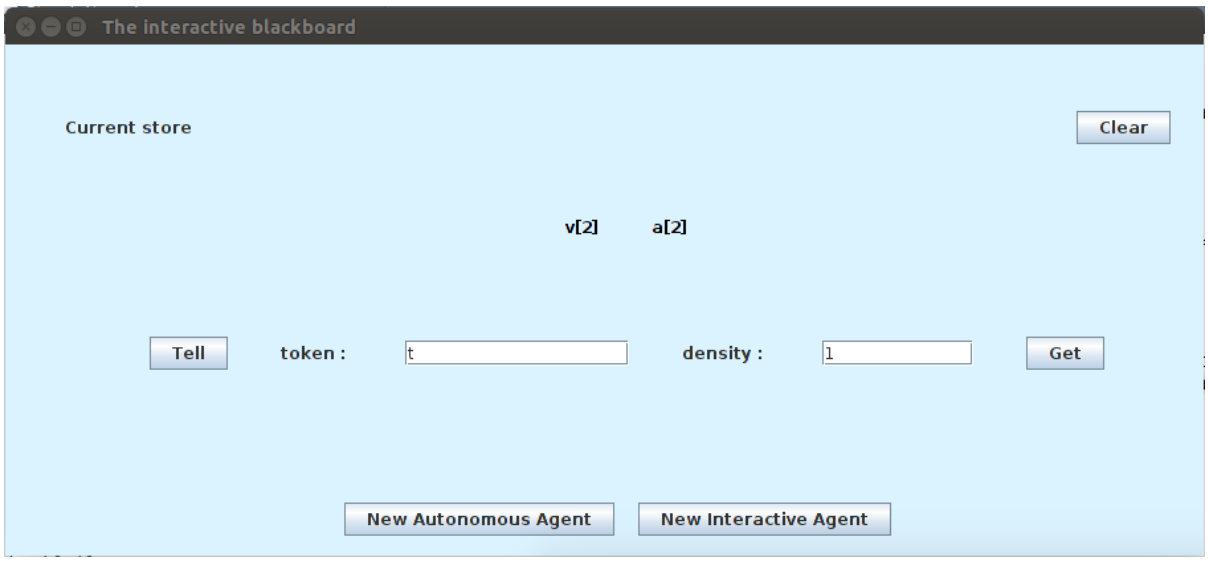


Figure 10.12: The first resulting `store` window

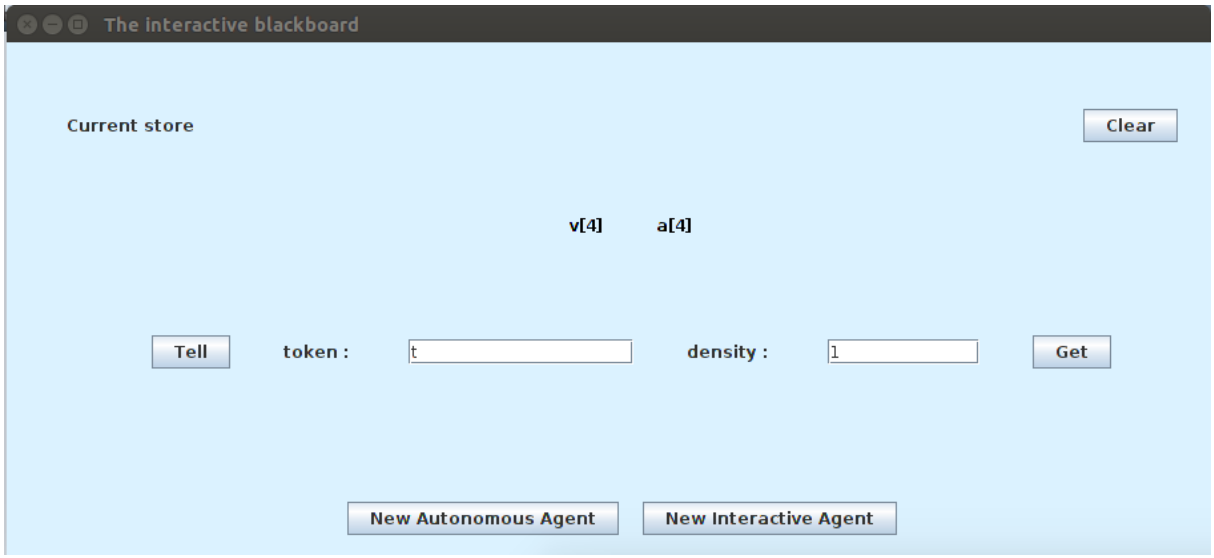


Figure 10.13: The possible second resulting `store` window

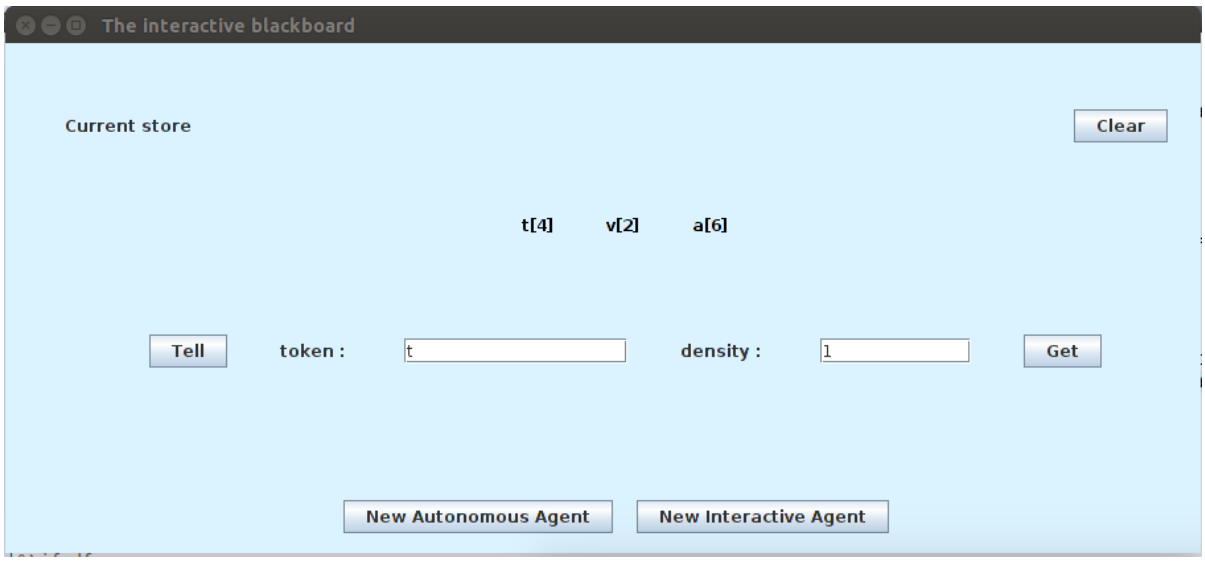


Figure 10.14: The possible third resulting `store` window

Pressing the "Next" button produces the same results step by step, again without control of the user on the choices to be made during the computation.

10.2 Implementation

10.2.1 Introduction

The graphical simulator is coded so as to meet the two computing possibilities offered hereabove to execute a Dense Bach agent. On the one hand, the user can let the program execute automatically the agent. On the other hand, the program parses the agent and, based on the state of the store, it offers to the user a choice of executable primitives. Moreover as presented in Section 10.1.2, the results of the parsing, the choice of the executable primitives and the state of the store must be visible in dedicated windows. Concretely the code is divided in seven parts : an abstract representation of the data, a parser, a class grouping the primitives acting on the store, a simulator that executes the parsed agent, and finally three classes that manage the three windows of the simulation, namely the window representing the store, the window for the automatic execution of the agent and the window for the interactive execution of the agent.

10.2.2 The structure of the data

The struture of the data is almost the same as the one used for the Dense Bach language, in the development of its interpreter and its command line simulator. Regarding the empty agent or the composed agents, the abstract class *Expr* is refined respectively in a class *DB_AST_Empty_Agent* or a class *DB_AST_Agent*. Regarding the primitives, a distinction must be done between those that are executable and those that are not, with respect to the state of the store. Two classes are then associated with the primitives: on the one hand, the *DB_AST_Primitive* for a primitive that is not executable with regard to the state of the store, and, on the other hand, the *DB_Exec_AST_Primitive* for a primitive that is executable. These two classes differ only by a parameter called *path*, only present in the class *DB_Exec_AST_Primitive*. In a form of a list of integers, it indicates the coordinates of the executable primitive inside the agent. The code of the class *Expr* is as follows:

```
class Expr
case class DB_AST_Empty_Agent() extends Expr
case class DB_AST_Primitive(primitive: String, token: String, density: Int)
                                                    extends Expr
case class DB_Exec_AST_Primitive(primitive: String, token: String, density: Int,
```

```
                                                    path:List[Int]) extends Expr
case class DB_AST_Agent(op: String, primitive: Expr, agent: Expr) extends Expr
```

10.2.3 The parser

The parser is exactly the one that has been developed for the interpreter and the command line simulator of the Dense Bach language and we thus refer the reader to the explanations of Subsection 8.3.2 of Chapter 8, and the code in annex (see Section F.2 of Chapter F). A class *PrettyPrinter* offers a function *translate* to transform a parsed agent *db_ag* in a human readable string. This class is defined inductively on the structure of *db_ag*. Concretely an empty agent is just transformed in an empty string. A primitive is written in its classical form : its name followed within parenthesis by the name of the token, that is itself followed by its density also within parenthesis. Finally for a composed agent, the function *translate* is called recursively on both subagents, that are written within square brackets, with the operator written in between. The code of the function *translate* is as follows:

```
class PrettyPrinter {
   def translate(db_ag: Expr):String = {
      db_ag match {
         case DB_AST_Empty_Agent() => ""
         case DB_AST_Primitive(db_prim,token,density) =>
             db_prim + "(" + token + "(" + density.toString + "))"
         case DB_AST_Agent(op,ag_i,ag_ii) =>
            "[ " + translate(ag_i) + " ] " + op + " [ " + translate(ag_ii) + " ]"
     }
   }
}
```

10.2.4 The store

The class *DBStore* is implemented as for the Dense Bach interpreter and command line simulator, in a form of a mapping that associates a token with a number representing the number of its occurrences on the store. The code of the four primitives *tell*, *ask*, *get* and *nask* is unchanged. In the case of an agent formed by a choice, subagents are selected based on their first step primitives, that must be checked as executable or not. Four boolean test functions *test_tell*, *test_ask*, *test_get* and *test_nask* perform this operation by checking the same conditions for their execution as the four primitives functions *tell*, *ask*, *get* and *nask*. If successfully selected they perform their action on the contents of the store. Finally, two functions *print_store* and *clear_store* permit to respectively print the store and to resset it to the empty map. The code

of the class *DBStore* is available in Section F.3 of Chapter F.

10.2.5 The simulator

The possibilities to execute an agent in an automatic or an interactive way imply a different processing of the parsed agent. The class *DBSimulExec* groups the functions invoked in both cases. The automatic processing relies on two functions called *run_one* and *exec_primitive*. The interactive processing uses four functions called *run_unselect*, *run_selected*, *test_exec_primitive* and *ag_first_steps*. We detail these functions hereafter.

The function *run_one* is defined as in Section 8.3.4, in an inductive way with regard to the structure of an expression *db_ag* of a parsed agent. It performs a transition step with respect to the operational semantics of Section 4.1.2. In case of the execution of a primitive, the function *run_one* makes use of function *test_exec_primitive*, that calls the associated primitive function on the store. For the parallel or choice composition, the first subagent to be executed is still selected randomly. Both subagents must be evaluated in case of the parallel composition, while only the selected one is evaluated for the choice composition. The complete code of functions *run_one* and *test_exec_primitive* is available in annex (see Section F.4 of Chapter F).

The interactive processing of an agent is based on the detection, at every step of the execution, of the executable primitives. These primitives are presented to the user as active buttons, among which he makes his choice. This approach needs to associate a coordinate with every primitive inside the agent. Based on the binary tree structure of the parsed agent, integer 1 is associated with the left branch, and integer 2 with the right branch, and this for every level of the tree. These numbers are added in a list of integer, the empty list being the coordinate of the root of the tree, corresponding to the initial agent.

Let us clarify this with an example and suppose for instance that the agent to be executed is as follows, starting from the empty store:

$$(tell(t(2)) \ ; ask(t(2))) \ || \ (nask(s(2)) \ ; get(t(2)))$$

It is clear that in this expression only the primitives *tell(t(2))* and *nask(s(2))* are executable. The empty list being associated with the global parallel agent, the subagents $tell(t(2)) \ ; ask(t(2))$ and $nask(s(2)) \ ; get(t(2))$ receive respectively the number 1 and 2, that are added in two lists [1] and [2]. Inside these lists, the primitives *tell(t(2))* and *ask(t(2))*, connected by a sequential operator, receive also the number 1 and 2 respectively. As the list of their composition is [1],

they have finally the respective coordinates [1,1] and [2,1]. In a similar way, the primitives *nask(s(2))* and *get(t(2))* receive respectively the coordinates [1,2] and [2,2].

The function *ag_first_steps* is defined inductively on the structure of its first argument *db_ag*, a parsed expression *Expr*. In particular for a primitive *DB_AST_Primitive* that is tested executable with the function *test_exec_primitive* on a given store, the function *ag_first_steps* returns an executable primitive *DB_Exec_AST_Primitive* with its *path* coordinate. In the contrary, the function just returns the primitive without any *path* coordinate. For a sequential composition, the function is called recursively on the first subagent, with a *path* list increased with 1. For the parallel and the choice compositions, the function acts recursively on both subagents ag_i and ag_{ii}, increasing their list of coordinates with respectively by 1 and 2. The complete code of *ag_first_steps* is as follows:

```
def ag_first_steps(db_ag: Expr,path:List[Int]):Expr = {
    db_ag match {
       case DB_AST_Empty_Agent() => DB_AST_Empty_Agent()

       case DB_AST_Primitive(db_prim,token,density) =>
          {  if (test_exec_primitive(db_prim,token,density,bb))
                              { DB_Exec_AST_Primitive(db_prim,token,density,path) }
             else { DB_AST_Primitive(db_prim,token,density) }
          }

       case DB_AST_Agent(";",ag_i,ag_ii) => DB_AST_Agent(";",
                                        ag_first_steps(ag_i,path:::List(1)),ag_ii)

       case DB_AST_Agent("||",ag_i,ag_ii) =>  DB_AST_Agent("||",
          ag_first_steps(ag_i,path:::List(1)),ag_first_steps(ag_ii,path:::List(2)))

       case DB_AST_Agent("+",ag_i,ag_ii) =>  DB_AST_Agent("+",
          ag_first_steps(ag_i,path:::List(1)),ag_first_steps(ag_ii,path:::List(2)))
    }
}
```

The coordinates being attributed to the executable primitives, the function *run_selected* performs the execution of a parsed agent, that is selected by the user. The function is defined inductively on the parsed expression *db_ag*. An empty agent stay unchanged. For a primitive, a distinction is made if the user selects a primitive that is executable or not. In the first case, the expression *DB_Exec_AST_Primitive* is executed with the function *exec_primitive* and the store is adapted. In the second case, the *DB_AST_Primitive* is returned unchanged, with no

action on the store. If the user selects a sequential composition *DB_AST_Agent("; ",ag_i,ag_ii)*, the function *run_selected* is applied on the first agent *ag_i*. If the resulting agent *new_ag* is empty, the second agent *ag_ii* is returned. Otherwise, the sequentially composed expression *DB_AST_Agent("; ",new_ag,ag_ii)* is returned. For a choice composition *DB_AST_Agent("+ ",ag_i,ag_ii)* only one of the two subagents *ag_i* and *ag_ii* can be executed. Following the choice of the user, the function *run_selected* is called on the first agent *ag_i*, or the second agent *ag_ii*. For the parallel composition *DB_AST_Agent("|| ",ag_i,ag_ii)*, both subagents must be executed. If the coordinate of the selected subagent is in the left part of the parallel agent, a *new_ag_i* is calculated with a *run_selected* call on the subagent *ag_i*. The second subagent *ag_ii* is processed with a function *run_unselect*, that transforms or maintains it in a not executable *new_ag_ii*. If the *new_ag_i* is empty, the parallel processing returns *new_ag_ii* as result, otherwise it returns a parallel composed agent *DB_AST_Agent("|| ",new_ag_i,ag_ii)*. The process is similar if the selected subagent is in the second branch of the composed agent.

The function *run_unselect* that is used in the process of a parallel composed agent is also defined in an inductive way, on a parsed expression *db_ag*. In case of a *DB_AST_Primitive*, the function maintains it in an unselected form if the primitive is not executable. It transforms it in an unselected form if the primitive is executable, but has not been choosen by the user. The complete codes of the functions *run_selected* and *run_unselect* are displayed hereafter:

```
def run_selected(db_ag: Expr,path:List[Int]): Expr = {

      db_ag match {

         case DB_AST_Empty_Agent() => DB_AST_Empty_Agent()

         case DB_AST_Primitive(db_prim,token,density) =>
                          DB_AST_Primitive(db_prim,token,density)

         case DB_Exec_AST_Primitive(db_prim,token,density,pp) => {
              exec_primitive(db_prim,token,density,bb)
              DB_AST_Empty_Agent() }

         case DB_AST_Agent(";",ag_i,ag_ii) =>  {
             val new_ag = run_selected(ag_i,path.tail)
             if ( new_ag == DB_AST_Empty_Agent() ) { ag_ii }
             else { DB_AST_Agent(";",new_ag,ag_ii) }
         }

         case DB_AST_Agent("||",ag_i,ag_ii) => {
```

```
            if ( path.head == 1 ) {
               val new_ag_i = run_selected(ag_i,path.tail)
               val new_ag_ii = run_unselect(ag_ii)
               if ( new_ag_i == DB_AST_Empty_Agent() ) { new_ag_ii }
               else { DB_AST_Agent("||",new_ag_i,new_ag_ii) } }
            else {
               val new_ag_i = run_unselect(ag_i)
               val new_ag_ii = run_selected(ag_ii,path.tail)
               if ( new_ag_ii == DB_AST_Empty_Agent() ) { new_ag_i }
               else { DB_AST_Agent("||",new_ag_i,new_ag_ii) } }
          }

         case DB_AST_Agent("+",ag_i,ag_ii) => {
            if ( path.head == 1 ) { run_selected(ag_i,path.tail) }
            else { run_selected(ag_ii,path.tail) }
          }
       }
   }

def run_unselect(db_ag: Expr): Expr = {

      db_ag match {

         case DB_AST_Empty_Agent() => DB_AST_Empty_Agent()

         case DB_AST_Primitive(db_prim,token,density) =>
                                       DB_AST_Primitive(db_prim,token,density)

         case DB_Exec_AST_Primitive(db_prim,token,density,pp) =>
                                       DB_AST_Primitive(db_prim,token,density)

         case DB_AST_Agent(op,ag_i,ag_ii) =>
                        DB_AST_Agent(op,run_unselect(ag_i),run_unselect(ag_ii))
       }
   }
```

The complete code of the class *DBSimulExec* is available in annex (see Chapter F in section F.4).

10.2.6 The interactive blackboard

The code of the window associated with the store is written as an executable object called *InteractiveBlackboard*, that extends the class *SimpleSwingApplication*. It is organised in two

parts, the first one for the representation of the current store, and the second one for the representation of the buttons that permit to invoke respectively, on the one hand, an automatic processing of the agent, and, on the other hand, an interactive processing of it. The windows associated with these two last parts are explained in Sections 10.2.7 and 10.2.8.

The representation of the store is defined as a fixed value called *theCurrentStore*, an instance of the *GridBagPanel* class. This means that all the elements that constitue the store are arranged in a grid with respects to constraints. The first element is the label *theStoreTitle*, initialized with the text "*Current store*". A set of constraints is associated with this label. The values *gridx* and *gridy* specify the grid cell where the element should be placed, in this case (0,0). The gridwidth specifies the number of grid cells that the element should span. The *weightx* parameter specifies how much the column should resize when the window is resized. In this case with a value of 0.5, there is almost no resizing. The *fill* parameter specifies how the element should fill its space in the layout. Here the default value *none* is used, that has been defined to the *Horizontal* one. This means that the component is wide enough to fill its display area horizontally. The parameter *anchor* is used to determine where to place the label inside the area, in this case in the west area. Among the other possible values are for instance *Center*, *North*, *NorthWest*. The last parameter *insets* is used to fix the minimum amount of space between the component and the edges of its display area. All these constraints are finally applied to the layout. The complete code of the label *theStoreTitle* is as follows:

```
val theStoreTitle = new Label { text = "Current store" }
c.weightx = 0.5
c.fill = Fill.None
c.gridx = 0
c.gridy = 0
c.gridwidth = 2
c.anchor = Anchor.West
c.insets = new Insets(5,5,5,5)
layout(theStoreTitle) = c
```

The value of the constraints can be redefined, if necessary, for every element to be placed within the grid. The next element is the clear button. It is defined as an instance of the class *Button*. It is initialized with the text "*Clear*"and is positioned at the East of the grid. Its code is as follows:

```
val theStoreClearButton = new Button { text = "Clear" }
c.anchor = Anchor.East
layout(theStoreClearButton) = c
```

An instance of the class *FlowPanel* is used to display the current contents of the store, defined as *bbObj*. This class arranges its contents horizontally, one after the other. The initial content is defined as a fixed label, with the text "*currently empty*". The values *hGap* and *vGap* specify the horizontal and vertical sizes of the panel. An empty transparent border is traced arround. The code of the *bbObj* is as follows:

```
val labelbb = new Label("currently empty")
val bbObj = new FlowPanel {
   background = blue
   opaque = true
   contents += labelbb
   hGap = 40
   vGap = 30
   border = Swing.EmptyBorder(15,10,10,10) }
c.gridx = 1
c.gridy = 1
layout(bbObj) = c
```

The store window offers to the user the possibility to insert dense tokens on the store or to retrieve them. This is done in order to adapt the store manually for permitting a complete execution of an agent. To do so another flowpanel, called *theStoreButtons*, is defined. It groups two buttons *tell* and *get*, and two text fields, the first one labelled with "*token*"to receive the name of the token to be inserted, and the second one labelled with "*density*"to receive its associated density. The token text field is initialized with *t*. The density text field is initialized with *1*. As they are defined in a flowpanel, all these elements are aligned horizontally. After their definitions as instances of the *Button*, *Label* or *TextField* classes, they are added to the content of the flowpanel *theStoreButtons*, according to the following code:

```
val theStoreButtons = new FlowPanel {
  background = blue
  val theTellButton = new Button { text = "Tell" }
  val theGetButton = new Button { text = "Get" }
  val theSTokenText = new Label { text = "token : " }
  val theSTokenField = new TextField { columns = 15
                                       text = "t" }
  val theSDensityText = new Label { text = "density : " }
  val theSDensityField = new TextField { columns = 10
                                         text = "1" }
  opaque = true
  contents += theTellButton
  contents += theSTokenText
```

```
  contents += theSTokenField
  contents += theSDensityText
  contents += theSDensityField
  contents += theGetButton
  hGap = 40
  vGap = 20
  border = Swing.EmptyBorder(5,10,5,10)
}
c.gridx = 1
c.gridy = 2
layout(theStoreButtons) = c
border = Swing.EmptyBorder(15,10,15,10)
```

Similarly to the *theCurrentStore* value, a fixed value *theCreateAgentButtons* is defined to group within a flowpanel the buttons used to invoke, on the one hand, an automatic execution of the agents, and, on the other hand, an interactive way of processing it. These two buttons receive as names, respectively "*New Autonomous Agent*", and "*New Interactive Agent*". The code to create this flowpanel is as follows:

```
val theCreateAgentButtons = new FlowPanel {
  val theCreateAutoAgentButton = new Button { text = "New Autonomous Agent" }
  val theStrutCreateButton = new Label { text = "  " }
  val theCreateInterAgentButton = new Button { text = "New Interactive Agent" }

  background = blue
  contents += theCreateAutoAgentButton
  contents += theStrutCreateButton
  contents += theCreateInterAgentButton
}
```

Both values *theCurrentStore* and *theCreateAgentButtons* being described, they have to be grouped in the main window. This the goal of the *top* function, that creates an instance of the class *MainFrame*. The two previous elements are to be placed vertically, and they are added vertically to the content of the main window, which is an instance of the class *BoxPanel*. The code of the function *top* is the following:

```
def top = new MainFrame {
    title = "The interactive blackboard"
    contents = new BoxPanel(Orientation.Vertical) {
         background = blue
         opaque = true
```

```
        contents += theCurrentStore
        contents += theStrutPanelI
        contents += theCreateAgentButtons
        border = Swing.EmptyBorder(30,30,10,10) }
}
```

All the buttons that have been defined previously must execute a specific action. As a reminder, these buttons are the *Clear* button, the *Tell* button, the *Get* button, the button *New Autonomous Agent* for processing automatically the agent, and finally the button *New Interactive Agent* for processing it in an interactive way. Specific functions are defined for these actions. They are respectively the *clear_store* function, the *tell_on_store* function, the *get_from_store* function, the *c_n_auto_agent* function and the *c_n_inter_agent* function.

The *clear* function calls the function defined with the same name in the class *DBStore*, that empties the store. Then the function *store_to_label* refreshes the element *bbObj* of *theCurrentStore*. The code of the *clear_function* is as follows:

```
def clear_store {
      mybb.clear_store
      store_to_label(mybb.theStore)
      println("Cleared the store")
   }
```

The code of the *store_to_label* function makes use of an internal function *newToken* to construct a new label associated with a pair (token, density). Clearing the current content of the *bbObj* element, the function *store_to_label* scans all the elements of the mapping *theStore*. The function *newToken* transforms them in labels that are added to the content of the element *bbObj* of *theCurrentStore*. The function *revalidate()* instructs the *layout manager* to recalculate the layout, as the current content has been cleared. The *repaint()* function repaints the component *bbObj* of *theCurrentStore*, as it has been modified. The code of the *store_to_label* function is as follows:

```
def store_to_label(theStore:Map[String,Int]) {

  def newToken(token:String,density:Int) = {
      new Label {  text = token + "[" + density.toString + "]"
           foreground = new java.awt.Color(0, 0, 0)
           background = jmjblue
           opaque = true }
  }
```

```
  theCurrentStore.bbObj.contents.clear
  for ((t,d) <- theStore)
         { theCurrentStore.bbObj.contents += newToken(t,d) }
  SwingUtilities.invokeLater(new Runnable(){
      public void run(){
          theCurrentStore.bbObj.revalidate()
          theCurrentStore.bbObj.repaint()
      }
}
```

The function *tell_on_store* provides the values inside the token field and the density field of *theCurrentStore* to the tell function of the class *DBStore*. When modified, the *store_to_label* function refreshes the content of the element *bbObj* of the *theCurrentStore*. The action of the *get_from_store* is similar, as it appears by comparison of both codes:

```
def tell_on_store {
    val token_arg = theCurrentStore.theStoreButtons.theSTokenField.text
    val density_arg = (theCurrentStore.theStoreButtons.theSDensityField.text).toInt
    val tres = mybb.tell(token_arg,density_arg)
    if (tres) {
         store_to_label(mybb.theStore)
         println("told " + token_arg + " with density " + density_arg)
         mybb.print_store
    }
}

def get_from_store {
    val token_arg = theCurrentStore.theStoreButtons.theSTokenField.text
    val density_arg = (theCurrentStore.theStoreButtons.theSDensityField.text).toInt
    val gres = mybb.get(token_arg,density_arg)
    if (gres) {
         store_to_label(mybb.theStore)
         println("got " + token_arg + " with density " + density_arg)
    }
}
```

The functions *c_n_auto_agent* and *c_n_inter_agent* create respectively an instance of the *InteractiveAutoAgent* and *InteractiveInterAgent* classes. These classes manage the windows for the automatic or interactive processing of the agents, respectively. As many windows of the same can be opened, they receive a different number at every creation. These classes are presented in Sections 10.2.7 and 10.2.8. The codes of these functions are as follows:

```
def c_n_auto_agent {
    nb_agent = nb_agent + 1
    val new_auto_agent = new InteractiveAutoAgent(nb_agent,mybb)
}

def c_n_inter_agent {
    nb_agent = nb_agent + 1
    val new_inter_agent = new InteractiveInterAgent(nb_agent,mybb)
}
```

All these functions must be associated with the proper button, to be executed when it is selected. The following code listens to the possible selection of the buttons, and associates with them the appropriate reaction.

```
listenTo( theCurrentStore.theStoreClearButton,
   theCurrentStore.theStoreButtons.theTellButton,
   theCurrentStore.theStoreButtons.theGetButton,
   theCreateAgentButtons.theCreateAutoAgentButton,
   theCreateAgentButtons.theCreateInterAgentButton )
   reactions += {
     case ButtonClicked(theCurrentStore.theStoreClearButton)
                          => clear_store
     case ButtonClicked(theCurrentStore.theStoreButtons.theTellButton)
                          => tell_on_store
     case ButtonClicked(theCurrentStore.theStoreButtons.theGetButton)
                          => get_from_store
     case ButtonClicked(theCreateAgentButtons.theCreateInterAgentButton)
                          => c_n_inter_agent
     case ButtonClicked(theCreateAgentButtons.theCreateAutoAgentButton)
                          => c_n_auto_agent
}
```

Finally, the five defined buttons are monitored. A click on one of them has for effect to call the appropriate function. This is represented in the following code:

```
listenTo( theCurrentStore.theStoreClearButton,
             theCurrentStore.theStoreButtons.theTellButton,
             theCurrentStore.theStoreButtons.theGetButton,
             theCreateAgentButtons.theCreateAutoAgentButton,
             theCreateAgentButtons.theCreateInterAgentButton )
   reactions += {
     case ButtonClicked(theCurrentStore.theStoreClearButton)
                                          => jmj_clear_store
```

```
case ButtonClicked(theCurrentStore.theStoreButtons.theTellButton)
                                        => jmj_tell_on_store
case ButtonClicked(theCurrentStore.theStoreButtons.theGetButton)
                                        => jmj_get_from_store
case ButtonClicked(theCreateAgentButtons.theCreateInterAgentButton)
                                        => c_n_inter_agent
case ButtonClicked(theCreateAgentButtons.theCreateAutoAgentButton)
                                        => c_n_auto_agent }
```

The complete code of the *InteractiveBlackboard* is available in annex (see Section F.5 of Chapter F).

10.2.7 The interactive execution

The class *InteractiveInterAgent* is an extension of the *Frame* class and manages the interactive execution of a Dense Bach agent. On the one hand, the class contains the code for constructing the window. On the other hand, the class provides the functions associated with the buttons of the window, to perform some specific actions. Within this window a text area serves to edit a Dense Bach agent. The parsing of this agent transforms it in a form where all the executable primitives are active buttons, that can be selected by the user. The class *InteractiveInterAgent* has for parameters the number of the agent, and the store.

Regarding the design of the window, it is constructed in a similar way as for the *InteractiveBlackboard* and is thus not presented in details. The complete code is available in Section F.6 of Chapter F. The different elements are positioned inside an instance of the class *GridBagPanel*, called *theAgent*. It contains essentially a scrollable agent field to edit the agent to be executed. A *Submit* button is available for parsing the agent. The result of the parsing is displayed in a flowpanel called *theCurrentAgentField*. In this panel, the syntax of the agent is preserved. Nevertheless all the primitives that it contains and that are executable with regard to the state of the store, are transformed in a button. Finally a *Refresh* button permits to adapt the executable state of the primitives in the parsed agent, taking into account any modification brought to the store in the *InteractiveBlackboard* window.

The function for the parsing of the agent to be parsed takes the text of *theAgentField* of *theAgent* element. It constructs a parsed agent by invoking the function *parse_agent* of the class *DBSimulParser*, on *the_agent_to_be_parsed*. Then the function *ag_first_steps* of the class *DBSimulExec*, acting on the *agent_parsed* and an empty list, constructs a *current_agent* expression, attributing to every primitive inside the agent a coordinate in the form of a list of integers. This expression is translated in a syntaxical form where every executable primitive is

represented by a button. Again both functions *revalidate()* and *repaint()* force the layout to be recalculated and the component *theCurrentAgentField* to be repainted. The complete code of the *parse_agent* function is as follows:

```
def parse_agent {
    agent_to_be_parsed = theAgent.theAgentField.text
    agent_parsed = mySimulParser.parse_agent(agent_to_be_parsed)
    current_agent = myDBsimul.ag_first_steps(agent_parsed,List())
    theAgent.theCurrentAgentField.contents.clear
    translate(current_agent)
    theAgent.theCurrentAgentField.revalidate()
    theAgent.theCurrentAgentField.repaint()
    println("output : "+agent_parsed)
    println("output agent labelled : "+current_agent)
}
```

The function *translate* has a *db_ag* expression as parameter and transforms it in a list of widgets. To perform its action, it uses essentially two functions: the *translate_into_widget_list* function, and the *group_db_widgets* function. The function *translate_into_widget_list* produces a list of *dbWidget*, that can be of two forms, a label or a button, as it results from the code of the following abstract class *dbWidget*:

```
class dbWidget
 case class DB_Label(txt_label: String) extends dbWidget
 case class DB_Button(txt_ag: String,path:List[Int]) extends dbWidget
```

The code of the function *translate_into_widget_list* is defined inductively on the structure of an *ag* expression. It makes essentially the difference between a *DB_AST_Primitive* and a *DB_Exec_AST_Primitive*. In the first case, a label is produced as widget. In the second case, a button is produced, with its coordinates, represented by *path*, inside the structure. The code of the function *translate_into_widget_list* is as follows:

```
def translate_into_widget_list(ag: Expr): List[dbWidget] = {
    ag match {
       case DB_AST_Empty_Agent() => List( DB_Label("Empty agent") )

       case DB_AST_Primitive(db_prim,token,density) =>  {
         val ag_label = db_prim + "(" + token + "," + density.toString + ")"
         List( DB_Label(ag_label) ) }

       case DB_Exec_AST_Primitive(db_prim,token,density,path) =>  {
```

```
          val ag_label = db_prim + "(" + token + "," + density.toString + ")"
          List( DB_Button(ag_label,path) ) }

        case DB_AST_Agent(op,ag_i,ag_ii) => {
          List( DB_Label( "[ " ) ) :::
          translate_into_widget_list(ag_i) :::
          List( DB_Label( " ] " + op + " [ " ) ) :::
          translate_into_widget_list(ag_ii) :::
          List( DB_Label( " ]" ) ) }
    }
}
```

The function *group_db_widgets* uses the list of dbWidgets produced by *translate_into_widget_list* to associate a button with the widgets of type *DB_Button*. If clicked by the user, this button performs one step in the execution of the agent. The code of the function *group_db_widgets* is defined in an inductive way on the list of dbWidgets, and is as follows:

```
def group_db_widgets(l_db_wi: List[dbWidget]) {
    l_db_wi match {

      case List() => { }

      case DB_Button(txt_ag,path) :: l_res =>  {
           theAgent.theCurrentAgentField.contents += newStepButton(txt_ag,path)
              group_db_widgets(l_res) }

      case DB_Label(txt_label) :: lres => {
           val (gen_label,ll_db_wi) = group_db_labels(txt_label,lres)
           theAgent.theCurrentAgentField.contents += newAgTxt(gen_label)
           group_db_widgets(ll_db_wi) }
         }
      }
```

In the case of *DB_Button* the function *newStepButton* creates an instance of a subclass *InteractiveStepButton*, as shown by the following code:

```
def newStepButton(prim_txt: String,path:List[Int]) = {
           new InteractiveStepButton(prim_txt,path)
      }
```

The subclass *InteractiveStepButton* extends the *Button* class. Based on its two parameters *prim_txt* and *path*, it defines the reaction of the associated button with the primitive *prim_txt*,

by invoking the function *execute_step* on *path*. The code of the subclass *InteractiveStepButton* is as follows:

```
class InteractiveStepButton(prim_txt:String,path:List[Int]) extends Button {
   this.text = prim_txt
   reactions += {
     case ButtonClicked(b) => execute_step(path) }
   }
```

The invoked function *execute_step* has for parameter the *path* coordinate of the button. With the function *run_selected* of the class *DBSimulExec*, it processes the selected primitive inside the *current_agent* and produces a resulting agent *res_agent*. Then based on it the function *ag_first_steps* determines a new current agent, with its executable primitives and their coordinates. The translation in executable buttons is finally displayed again in the current agent field, with the actions of *revalidate()* and *repaint()*. The code of the function *execute_step* is as follows:

```
def execute_step(path:List[Int]) {
    res_agent = myDBsimul.run_selected(current_agent,path)
    current_agent = myDBsimul.ag_first_steps(res_agent,List())
    theAgent.theCurrentAgentField.contents.clear
    translate(current_agent)
    theAgent.theCurrentAgentField.revalidate()
    theAgent.theCurrentAgentField.repaint()
    println("new agent : "+res_agent)
    InteractiveBlackboard.redisplay_store
   }
```

If the agent evolves to a non empty form, with no possibility to make a next step, the user can adapt the store of the *InteractiveBlackboard* with the *Tell* and *Get* buttons, as explained in Section 10.2.6. Clicking on the button *Refresh* forces a new parsing of the resulting agent *res_agent*, to take into account the new configuration of the adapted store. Concretely this is done by using the functions *ag_first_steps* and *translate* to determine what are the executable primitives inside the agent, and to present them as buttons to be clicked by the user. The functions *revalidate()* and *repaint()* make the new agent visible as new content. The code of the function *refresh_agent* is as follows:

```
def refresh_agent {
    current_agent = myDBsimul.ag_first_steps(res_agent,List())
    theAgent.theCurrentAgentField.contents.clear
```

```
    translate(current_agent)
    theAgent.theCurrentAgentField.revalidate()
    theAgent.theCurrentAgentField.repaint()
}
```

Finally, the functions *parse_agent* and *refresh_agent* must be associated with their appropriate buttons, as reactions to be triggered in case of their clicking. The code of this attribution is follows:

```
listenTo(theAgent.theSubmitAgentButton,theAgent.theRefreshAgentButton)
reactions += {
  case ButtonClicked(theAgent.theSubmitAgentButton) => parse_agent
  case ButtonClicked(theAgent.theRefreshAgentButton) => refresh_agent
```

The complet code of the class *InteractiveInterAgent* is available in annex (see Section F.6 of Chapter F).

10.2.8 The automatic execution

The user has for second option to execute the Dense Bach agent, on the one hand, in an automatic way or, on the other hand, in a step by step way. The class that manages this option is called *InteractiveAutoAgent*, which is an extension of the *Frame* class. This class has for parameters the number of the agent, and the store, and produces the window for introducing the agent to be processed. The text field *theAgentField* to edit the Dense Bach agent, and the current agent field that displays the result of its parsing in a human readable syntax are inserted in a instance of a *GridBagPanel*, in a similar way as for the windows of the *InteractiveBlackboard* and the *InteractiveInterAgent* classes. It is not presented in details but is avalaible in annex (see Section F.7 of Chapter F).

Three buttons are displayed in the window. They are respectively the *Submit* button, the *Run* button and the *Next* button. They are associated with three functions that precise the reaction to be triggered if the user chooses to click one of them. The function *parse_agent* is associated with the button *Submit*. This function parses the Dense Bach agent edited in the *theAgentField* text area. It invokes the function *parse_agent* of the class *DBSimulParser* to parse the Dense Bach agent. Then it uses the function *translate* of the class PrettyPrinter, to display the label *theCurrentAgentField* in a human readable text format. The code of the function *parse_agent* is as follows:

```
def parse_agent {
    agent_to_be_parsed = theAgent.theAgentField.text
    agent_parsed = mySimulParser.parse_agent(agent_to_be_parsed)
    current_agent = agent_parsed
    theAgent.theCurrentAgentField.text = myTranslator.translate(current_agent)
    println("output : "+agent_parsed)
}
```

The second function *execute_step* is associated with the button *Next*. This function invokes the *run_one* function of the class *DBSimulExec* to process one step of execution of the Dense Bach agent. The result of the function is a boolean that indicates if the execution of the step is successful or not. In case of success, the resulting new agent is translated in a human readable way and redisplayed in the text area of *theCurrentAgentField*. The code of this function is as follows:

```
def execute_step:Boolean = {
    var exec_result = true
    previous_agent = current_agent
    if (current_agent != DB_AST_Empty_Agent()) {
      exec_result = myDBsimul.run_one(current_agent) match
           { case (false,_)          => false
             case (true,new_agent)  =>
               { current_agent = new_agent
                 true
               }
           }
      theAgent.theCurrentAgentField.text = myTranslator.translate(current_agent)
      InteractiveBlackboard.redisplay_store
    }
    exec_result
}
```

Finally the third function is the *execute_all* function, associated with the *Run* button. Under the condition that there is still an agent to process, and that no failure in the execution has occured, this function executes in a while loop the previous *execute_step* function. The code of this function is as follows:

```
def execute_all = {
    var failure = false
    while ( current_agent != DB_AST_Empty_Agent() && !failure ) {
       println("enter execution step")
```

```
        failure = !execute_step
    }
}
```

These three functions are associated with their respective buttons, as reactions in case of their selection, through the following code:

```
listenTo(theAgent.theSubmitAgentButton,
   theAgent.theRunningButtons.theStepAgentButton,
   theAgent.theRunningButtons.theRunAgentButton)
reactions += {
   case ButtonClicked(theAgent.theSubmitAgentButton) => parse_agent
   case ButtonClicked(theAgent.theRunningButtons.theStepAgentButton) => execute_step
   case ButtonClicked(theAgent.theRunningButtons.theRunAgentButton) => execute_all
}
```

The complete code of the class *InteractiveAutoAgent* is available in Section F.7 of Chapter F.

10.3 Living example

Let us illustrate the use of the simulator with an example and let us suppose that the user wants to observe the execution of the following Dense Bach agent:

$$(tell(a(3)) \ ; nask(u(2)) \ || \ get(r(2)))$$

Let us suppose that the user wants to study the execution in an interactive way. In the *Interactive Blackboard* window of Figure 10.15, he selects the button *New Interactive Agent*, that produces the window of Figure 10.16, where he can edit the agent $tell(a(3)) \ ; nask(u(2)) \ || \ get(r(2))$.

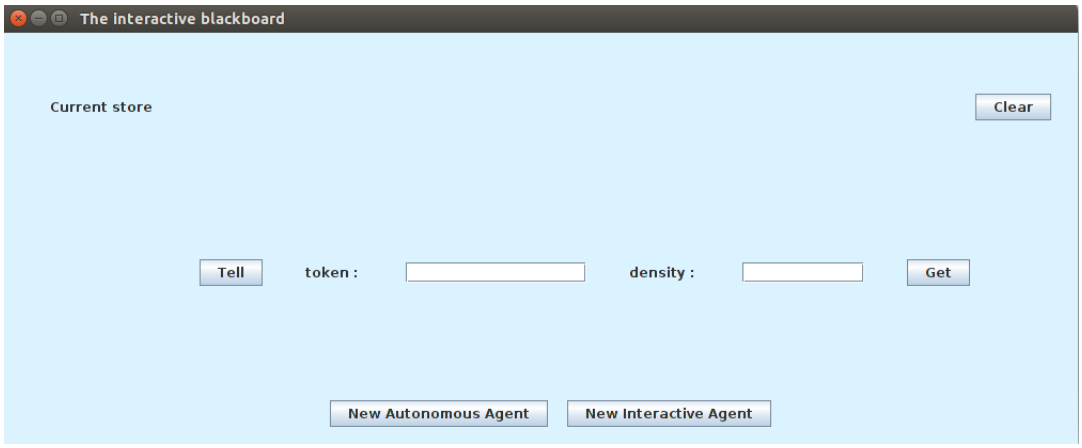


Figure 10.15: The `Interactive Blackboard` window with an empty store

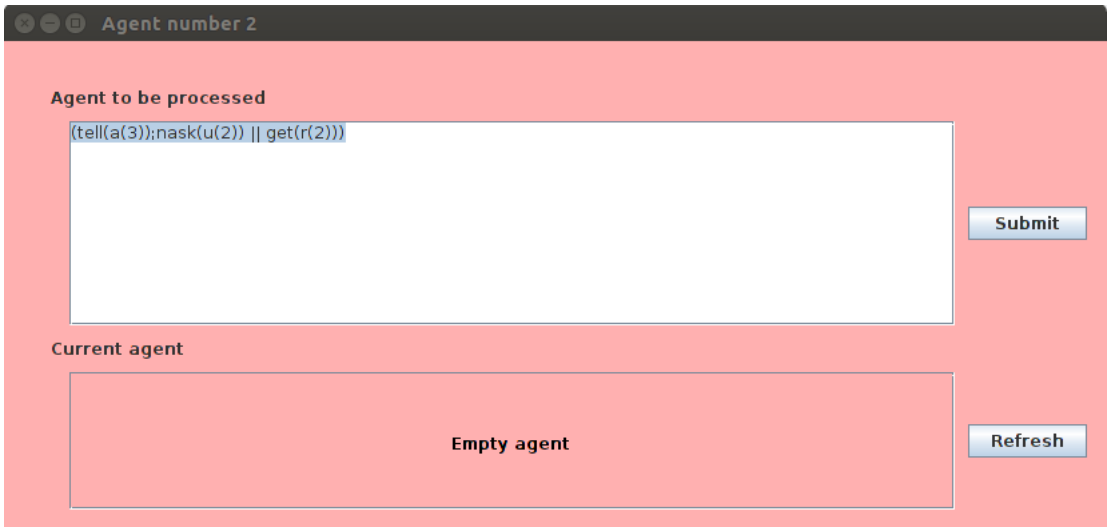


Figure 10.16: The `Interactive Agent` window with the agent edited

In Figure 10.17 the agent is parsed, proposing the primitive *tell(a(3))* to be executable.

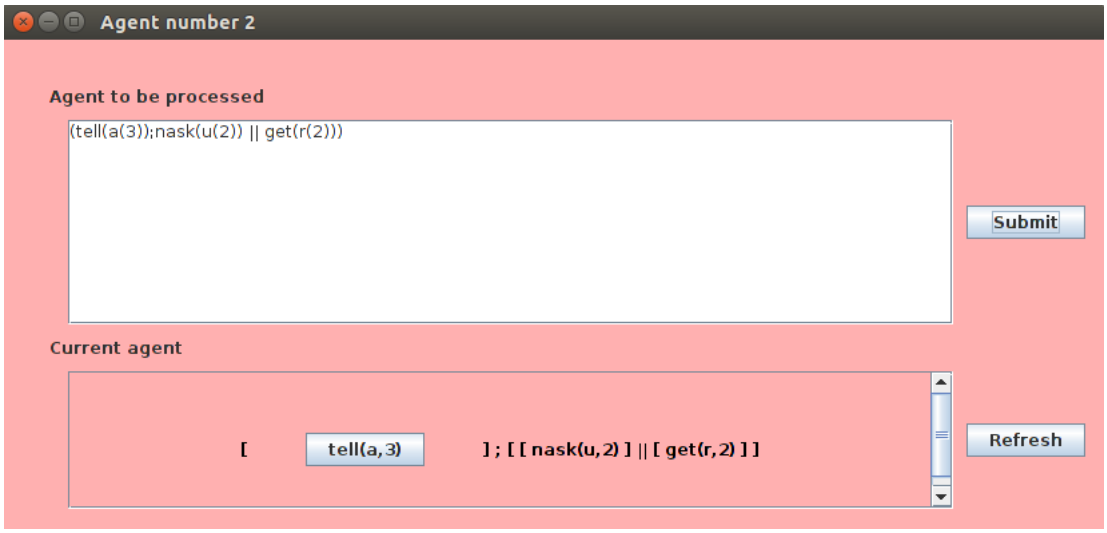


Figure 10.17: The parsed agent with the primitive `tell(a(3))` executable

The result of the execution of the primitive *tett(a(3))* introduces three tokens *a* in the store, as shown in Figure 10.18.

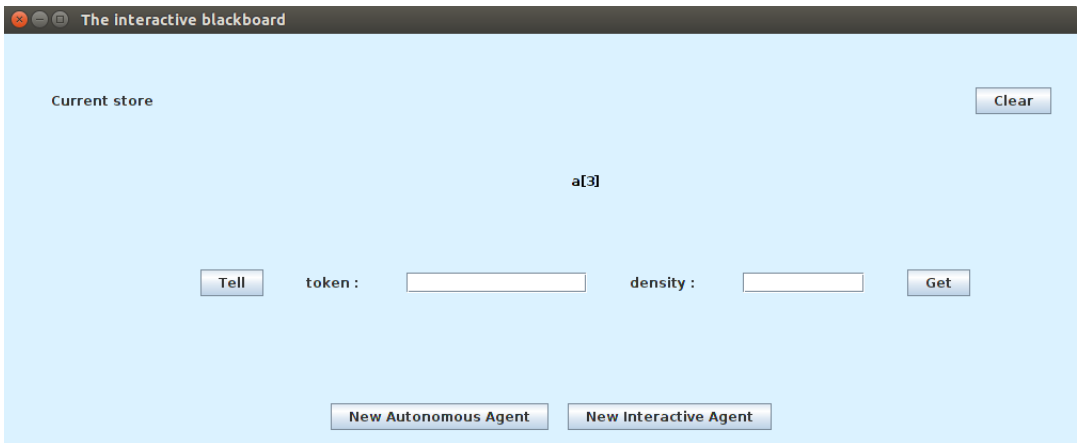


Figure 10.18: The store with the three tokens `a`

Figure 10.19 indicates that the primitive *nask(u(2))* is now executable.

Figure 10.19: The parsed agent with the primitive `nask(u(2))` executable

After the succesful execution of the *nask(u(2))* primitive, Figure 10.20 shows the resulting agent stopped on the *get(r(2))* primitive that cannot be executed, following the state of the store.

Figure 10.20: The parsed agent with the primitive `get(r(2))` non executable

In order to make the last primitive *get(t(2))* executable, the store is adapted with its *Tell* button, by introducing four instances of the token *r*. Figure 10.21 shows this adaptation.

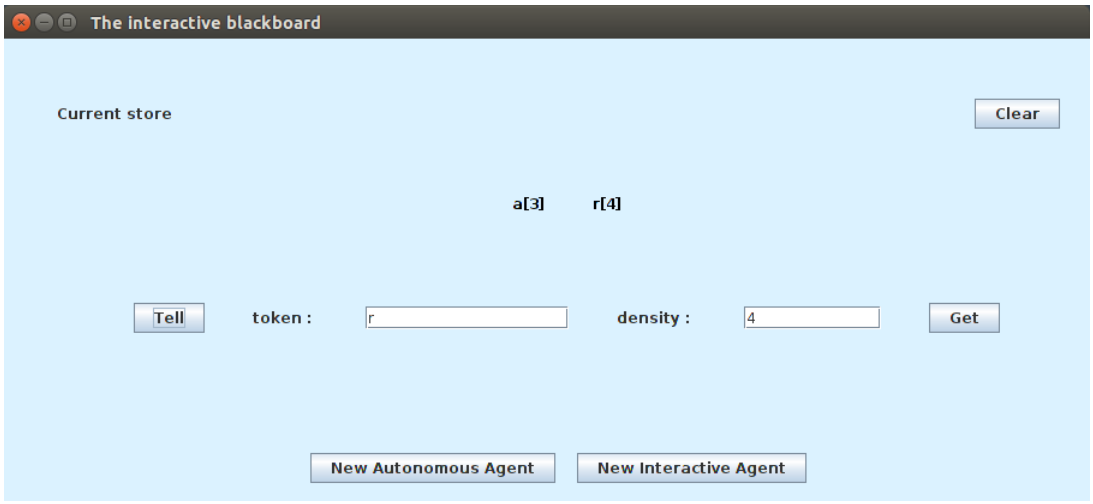


Figure 10.21: The store with the four tokens `r` added with the `Tell` button

With the new configuration of the store, the *get(r(2))* primitive can be activated. This is done with the button *Refresh* in the interactive window. Figure 10.22 shows this activation.

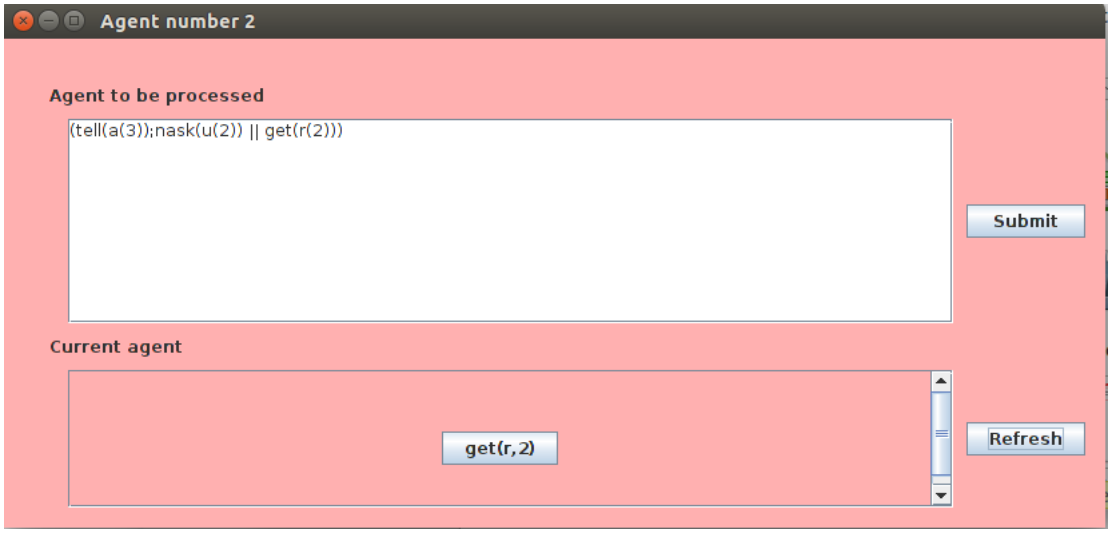


Figure 10.22: The `get(r(2))` activated with the refresh button

The last primitive *get(r(2))* is now executed. As a result the final resulting agent is empty, as shown in Figure 10.23, and the final store contains three instances of the token a and two of the token r, as shown in Figure 10.24.

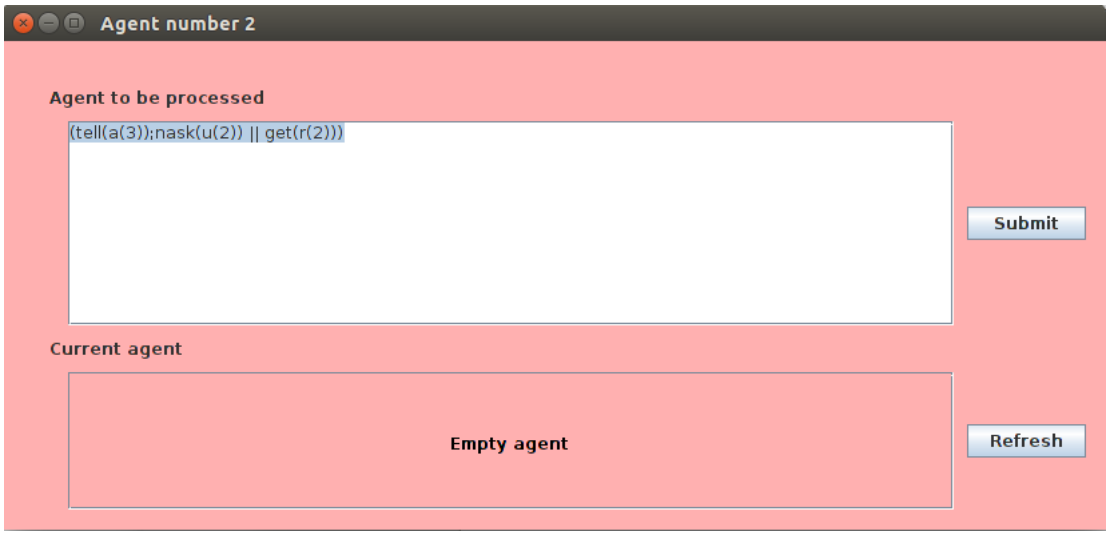


Figure 10.23: The empty agent after the execution of the `get(r(2))` primitive

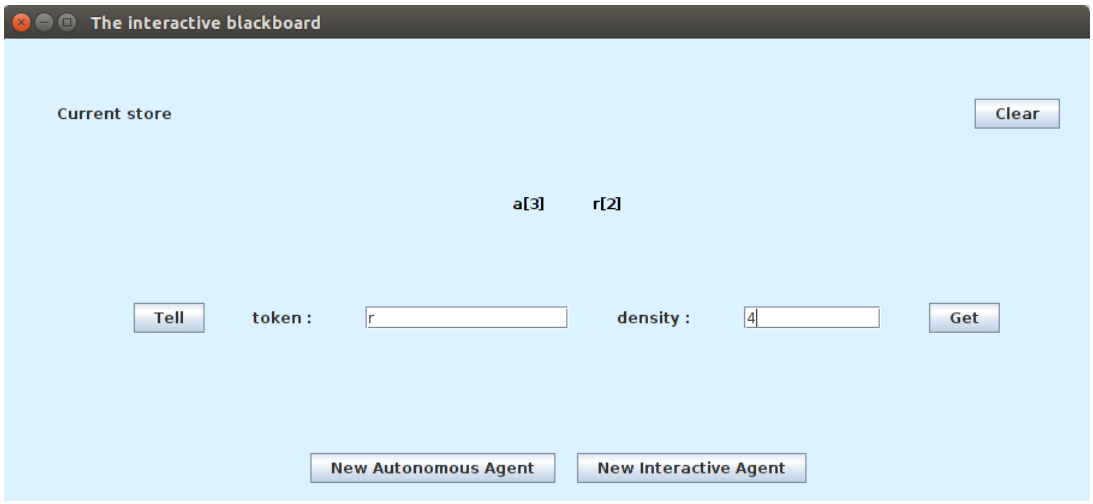


Figure 10.24: The final store with the two tokens `r` and the four tokens `a`

Let us suppose now that the user wants to execute the agent $tell(a(3))$; $nask(u(2))$ || $get(r(2))$ in an automatic way, starting from a store cleared into an empty state with its *Clear* button, reproducing the Figure 10.15. Figure 10.25 shows the window of the autonomous agent, obtained by clicking the button *New Autonomous Agent*. In this window the agent has been edited and parsed, with the button *Submit*.

As the execution of the agent with the *Run* button is done in the same conditions, the resulting agent stops again on the primitive *get(r(2))*, as shown in Figure 10.26. In the same

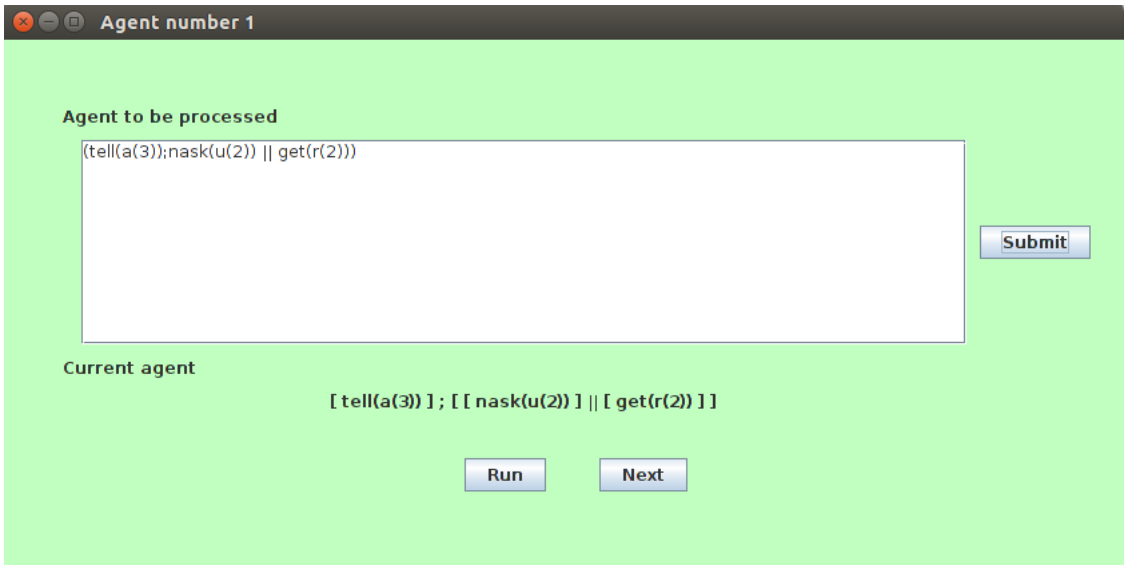


Figure 10.25: The `Autonomous Agent` window with the agent edited and parsed

time, three units of the token a are placed on the store, reproducing Figure 10.18.

Figure 10.26: The resulting agent after the `Run` execution

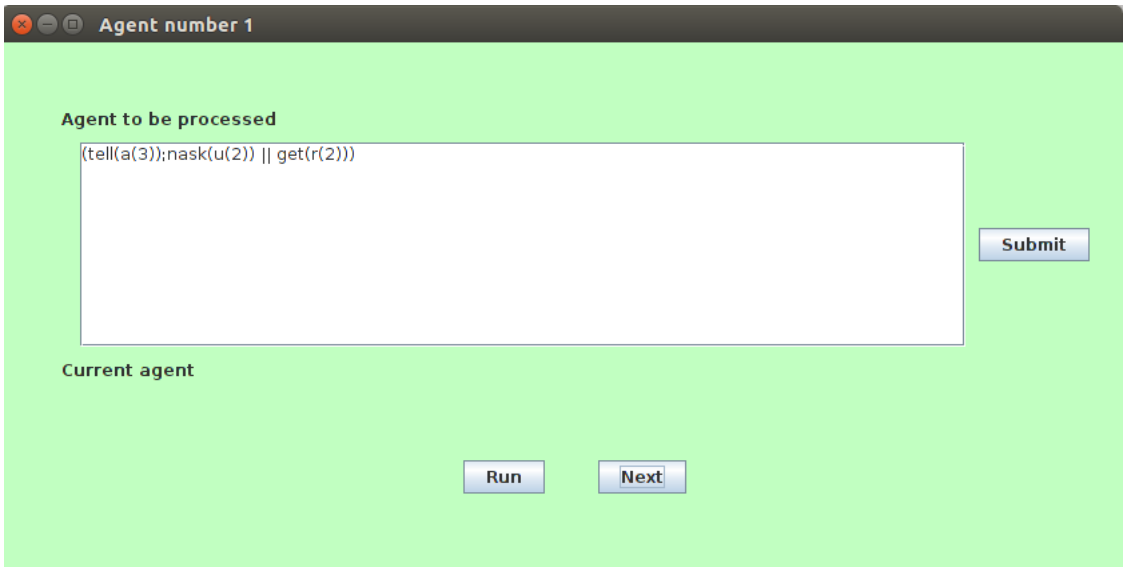


Figure 10.27: The autonomous agent after the `Step` execution

To solve this problem, the store is again adapted, by introducing four units of the token r, using the *Tell* button, reproducing the situation of Figure 10.21. The last primitive can now be executed, by using the button *Next*. Figure 10.27 shows an empty current agent after this successful execution, that has also for effect to retrieve two units of the token r on the store, leading to the same configuration as in Figure 10.24.

10.4 Conclusion

In this chapter, we have developed a simulator for the execution of Dense Bach agents. It offers the user three windows for visualizing the different ways to execute an agent, and the effect of its execution on the store. The executions can be done automatically, or step by step, or even in a complete interactive way, by providing the user the choice of the primitive that he wants to execute. By its graphical approach this tool completes the interpreters and command line simulators that have been developed.

Chapter 11

Modeling Dense Bach with Petri Nets

Programs of concurrent and distributed systems are complex, as they can combine lots of components and interactions. The analysis of the execution of such programs is at the same time also very complex. Programs written in the dense languages we have proposed are no exception to this situation. On the other hand, Petri Nets are widely recognized as good tools to analyze concurrent programs, as they provide in details all the possible states of the execution of a concurrent program and allow for the analysis of classical properties such as reachability and deadlock-freedom. Vandorpe [Van09] has developed a software to convert a Bach expression into a Petri net, to analyze the resulting net and to simulate, using this net, the execution of the initial Bach expression. Building upon this piece of work, we propose an extension to model and analyze Dense Bach agents. Another novelty with respect to [Van09] is that we use the concept of Open Petri Nets introduced in [BBG09]. This extension has been programmed with the Scala language. As for Chapter 10, we shall again concentrate on Dense Bach.

11.1 Open Petri nets

In [BBG09], Baldan has defined an open Petri net as an ordinary place / transition net endowed with distinguished sets of open places, representing the interfaces through which the environment interacts asynchronously with a net.

Definition 28. Denoting by $S^{\oplus}$ the free commutative monoid over S, the element of which are called multiset over S, an open (P/T Petri) net is a tuple $N = (S, T, {}^{\bullet}(.), (.)^{\bullet}, O)$ where S is the set of places, T is the set of transitions, ${}^{\bullet}(.)$,$(.)^{\bullet} : T \to S^{\oplus}$ are functions mapping each transition to its pre- and post-set, and $O \subseteq S$ is the set of open places.

Figure 11.1 represents an open Petri net. In this picture, open places carry a label and are represented by double circles. Other places are also marked, as are transitions as well. Referring to Definition 29 and assuming that pre-conditions and post-conditions of transitions only handle one token, the elements are as follows :

$$
\begin{aligned}
S &= \{a, b, c, P_1, P_2, P_3, P_4\} \\
T &= \{t_1, t_2, t_3, t_4\} \\
{}^\bullet(t_1) &= \{a, P_1\} \\
{}^\bullet(t_2) &= \{P_1\} \\
{}^\bullet(t_3) &= \{P_3, P_4\} \\
{}^\bullet(t_4) &= \{P_2, P_3\} \\
(t_1)^\bullet &= \{a, P_2\} \\
(t_2)^\bullet &= \{P_2, P_3\} \\
(t_3)^\bullet &= \{c, P_4\} \\
(t_4)^\bullet &= \{b\} \\
O &= \{a, b, c\}
\end{aligned}
$$

Figure 11.2 depicts a second example of an open Petri net, in the form of a duplicator agent. It consumes one token provided by the environment in place a and produces two token in places b and c.

11.2 DB-open Petri Nets

In his master thesis, Vandorpe [Van09] has proposed a model to translate Bach agents in Petri nets. In this model every BachT agent is associated with a compositional block, which schematically is composed of an entry transition, followed by intermediate places and transitions to finally reach an exit place. Figure 11.3 represents this basic block, where the oval rectangle corresponds to the compositional block. It has on its left part a transition t, and on its right part a P_{out} place, that receives the result of the firing of the agent. This place is reached if and only if all the actions of the primitive on the store have been completed successfully. The P_{in} place is not part of the compositional block, and serves to initialize the compositional block. This place corresponds to the P_{out} place of the compositional agent that may precede the one of figure 11.3.

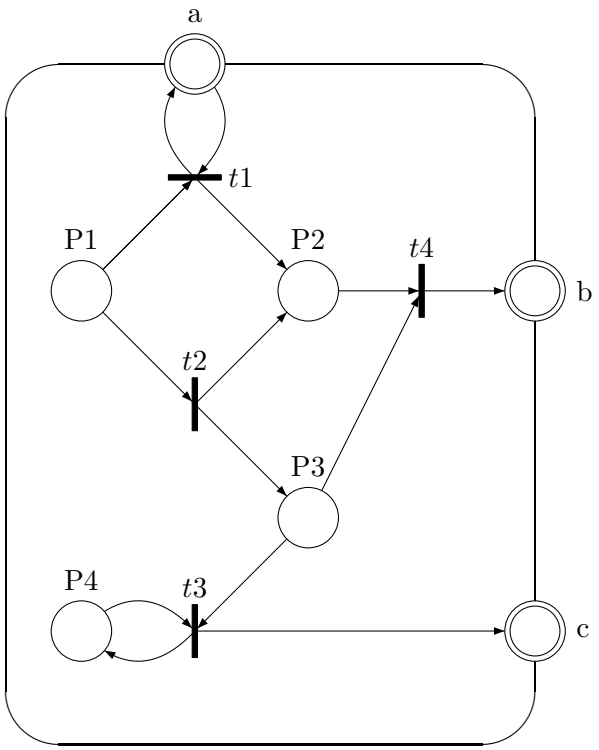


Figure 11.1: An example of an open Petri net (from [BBG09]).

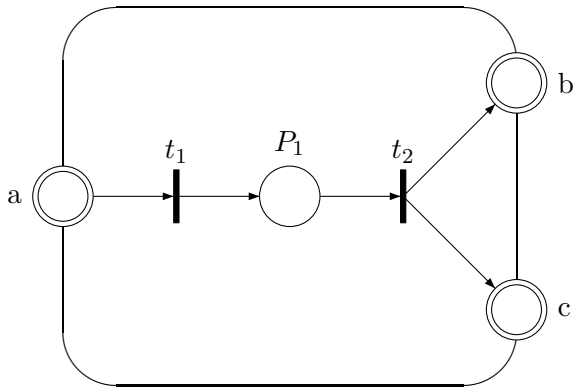


Figure 11.2: A duplicator agent in open Petri Net

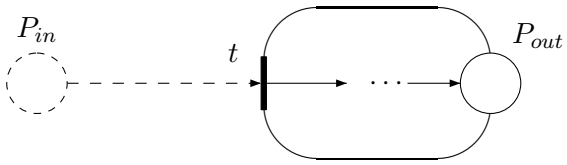


Figure 11.3: Basic block of the model of Vandorpe in a Petri net

In contrast, our idea is to express that block by using the open places of a Petri net. As a result, every agent is represented by two places, one for the entry place (P_{in}) and the other for the exit place (P_{out}). The entry place is potentially at the source of a finite number of transitions. Each of these transitions corresponds to a path of execution of the agent. The exit place is pointed by a finite number of transitions. The firing of each of them will produce one token that will be received by the exit place. As we will see later, the idea is that the first transitions may compete with each other to proceed but some of them may be executed. In contrast, only one of the final transition will be executed. It is also worth observing that the entry transitions may require more than one token to be consumed whereas the final transitions only produce one token. The fact that entry transitions may consume more than one token results in defining a number of tokens to be provided in the entry place to allow the agent to proceed. Hence, our translation not only provides a Petri Net but also such a number of tokens.

Figure 11.4 depicts our structure of an agent, with n transitions related to the entry place and m related to the exit place. In this figure, the places P_{in} and P_{out} are open places, and are the only places visible from the environment. They represent respectively the entry and exit places of the Petri Net associated with the considered agent. In particular the exit place indicates the end of the execution of the agent, and is always used as an activator of the next agent. The black rectangles represent the transitions (which are firable if their pre-condition is satisfied). The arrows from places to transitions are equipped with a weight that represents the number of tokens consumed by the firing of the transitons. In particular, the numbers α_1, α_2, ..., α_n are the tokens consumed by transitions t_1, t_2, ..., t_n, respectively. Similarly, the numbers on the arrows from transitions to places indicate the number of tokens produced in the

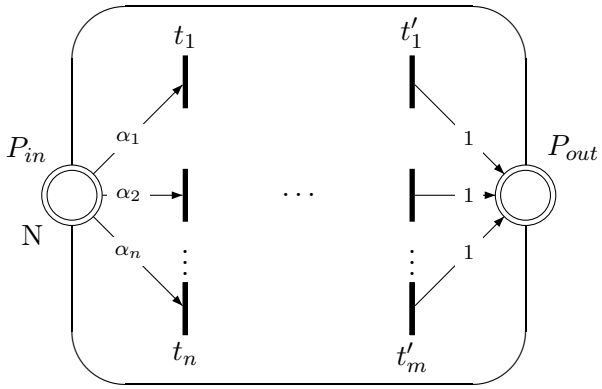


Figure 11.4: A compositional agent in Open Petri Net

places by firing the transitions. Finally, the number N indicates the initial number of tokens N assumed to be present in the place P_{in} to allow the Petri Net to mimick the behaviour of the corresponding Dense Bach agent.

To complete the picture, one has to take care of the representation of the store. Following [Van09], the content of the store is modeled by places in the corresponding Petri Net. Every token on the store is associated with two places. The presence of n tokens in a first place means that n instances of the token are on the store. Any interaction of a primitive with a token on the store consists then to connect the transition in the basic block of the primitive, with the place associated with the token. In the following, to make the distinction between the token manipulated by the Dense Bach language in the token space, and the token representing the flux of execution of the Petri net, we shall respectively name them Dense Bach token and Petri token.

The Dense Bach primitives *tell*, *ask*, *get* and *nask* are the only four possible interactions with the token space. The three first are easy to be modeled in a Petri net, as they only consist in adding or retrieving some Dense Bach tokens in the place representing the Dense Bach token they are acting on. However the fact that Petri Nets are not Turing complete induces a difficulty to represent the *nask* primitive. Indeed in absence of inhibitor arcs, the formalism of Petri cannot easily test the absence of a Dense Bach token in a place, as it cannot fire a transition starting from this empty place.

This difficulty is got arround by associating a second place with any Dense Bach token. More formally, we assume, for any dense token, a maximum number of occurrences, say MAX, and we then associate every Dense Bach token t present on the store with a pair of places P_t

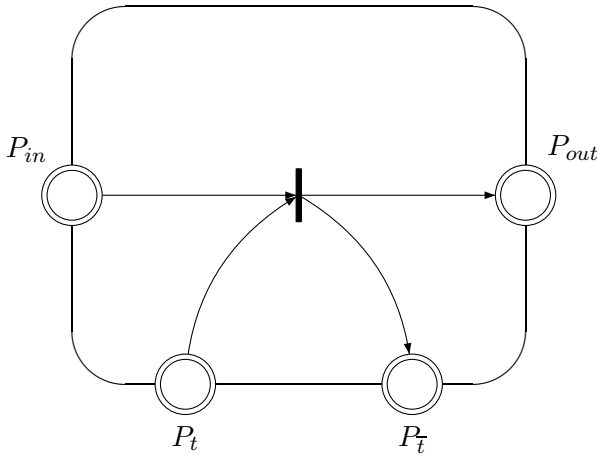


Figure 11.5: A compositional agent in Open Petri Net

and $P_{\overline{t}}$ in the token space. The place P_t contains the number of Petri tokens representing the number of instances of t that are present on the store. The $P_{\overline{t}}$ place, also called the *anti-place*, keeps the number of instances that are not yet used.

In our model of *place* and *anti-place*, the situation where there is no Dense Bach token t on the store corresponds on the one hand to an empty place P_t, and on the other hand to a $P_{\overline{t}}$ which contains the maximum number MAX of Petri tokens. Every primitive will have a specific effect on these places. Following their types, these effects are explained in subsection 11.3.1.

It is worth observing that the two places must obey an invariant property that we will guarentee subsequently : for any token t, the sum of the number of tokens in P_t and the number of tokens in $P_{\overline{t}}$ is equal to MAX. Note also that we might consider that MAX varies from token to token. However, for the ease of the presentation, we shall stick to a fixed number MAX for all the tokens.

Figure 11.5 represents one of the simplest open Petri net structures of a Dense Bach agent interacting with the representation of a dense token on the store. In this case it corresponds to *get(t(1))*. There the Dense Bach token t is associated with a pair of places P_t and $P_{\overline{t}}$. This leads us to the more general form illustrated in Figure 11.6. There, for the ease of the presentation, names of the transitions and weight of the final transitions, present in Figure 11.4, have been omitted and places and anti-places associated with the tokens under consideration are drawn without the outgoing and incoming arcs between them and the transitions.

As a conclusion for this section, we can formalize the Open Petri Nets we shall consider through the notion of DB-open Petri Net.

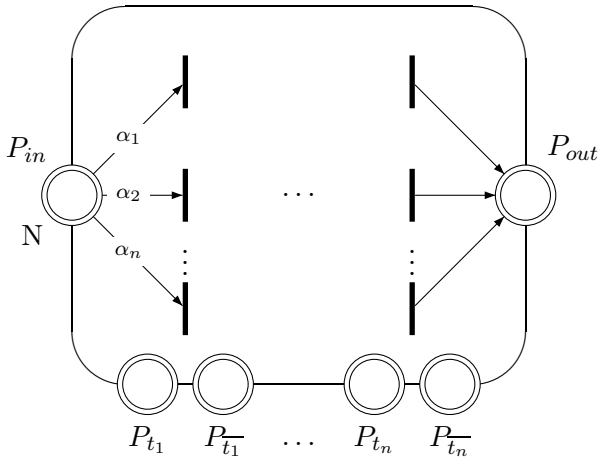


Figure 11.6: General form of a Petri Net associated with an agent

Definition 29. Given a set of tokens $\mathcal{T}$, a *DB-open Petri Net* on $\mathcal{T}$ is a tuple of the form $(S, T, {}^\bullet(.), (.)^\bullet, O, P_{in}, P_{out}, Pts, M, N)$ where

- S is the set of places of the Petri Net with
  - P_{in} being the entry place,
  - P_{out} being the exit place,
  - $Pts = \{P_t : t \in \mathcal{T}\} \cup \{P_{\overline{t}} : t \in \mathcal{T}\}$ being the set composed of the places and anti-places associated with the tokens in $\mathcal{T}$
  - $O = \{P_{in}, P_{out}\} \cup Pts \subseteq S$ being the set of open places
- T is the set of transitions with ${}^\bullet(.)$ and $(.)^\bullet : T \to S^\oplus$ being functions that map each transition to its pre- and post-set
- $M : S \to \mathbb{N}$ is a partial marking of places such that $M(P_t) = 0$ and $M(P_{\overline{t}}) = MAX$ for any $t \in \mathcal{T}$
- N is a (strictly) positive number

As the careful reader will have noticed, the marking M essentially provides the initialisation of the places and anti-places associated with the tokens. We shall however also employ it later to provide tokens to auxiliary places in order to control the execution of the Petri Net associated with a Dense Bach agent.

In the following, for the ease of the presentation, we shall subsequently abuse language and employ the term DB-open Petri Net on an agent A to denote a DB-open Petri Net on the set

of tokens contained in A. The precise definition of the DB-open Petri Net associated with an agent is provided inductively by the following section.

11.3 Modeling Dense Bach agents

11.3.1 The basic primitives

The following subsections present the translation in a DB-open Petri net of the four basic primitives of Dense Bach. Figures 11.7 to 11.10 constitute the basic bricks for the construction of any Petri net, corresponding to the translation of a Dense Bach expression. The form of the figures presents always the same structure: every net begins with an open place and must finish with an open place, both being represented by a double circle.

11.3.1.1 The tell primitive

Figure 11.7 depicts the representation of the tell primitive in an open Petri net. Basically, the introduction of a Dense Bach token *t(m)* with density m on the store triggers a transfer of m Petri tokens from the anti place $P_{\overline{t}}$ to the place P_t, as the direction of the arrows indicates. Regarding the Petri Net, the firing of the transition also consumes one Petri token in the entry open place P_{in}, and produces one Petri token in the exit open place P_{out}.

With the agent *tell(t(m))* we thus associate the following DB-open Petri Net $(S, T, {}^\bullet(.), (.)^\bullet, O, P_{in}, P_{out}, Pts, M, N)$ on $\mathcal{T} = \{t\}$:

- $S = O = \{ P_{in}, P_{out}, P_t, P_{\overline{t}}\}$
- $P_{in} \in O$ is the entry place
- $P_{out} \in O$ is the exit place
- $Pts = \{P_t\} \cup \{P_{\overline{t}}\}$
- $T = \{tell(t(m)) \}$
- ${}^\bullet(tell(t(m))) = \{P_{\overline{t}}(m), P_{in}\}$
- $(tell(t(m)))^\bullet = \{P_t(m), P_{out}\}$
- $M(P_t) = 0$ and $M(P_{\overline{t}}) = MAX$
- N = 1

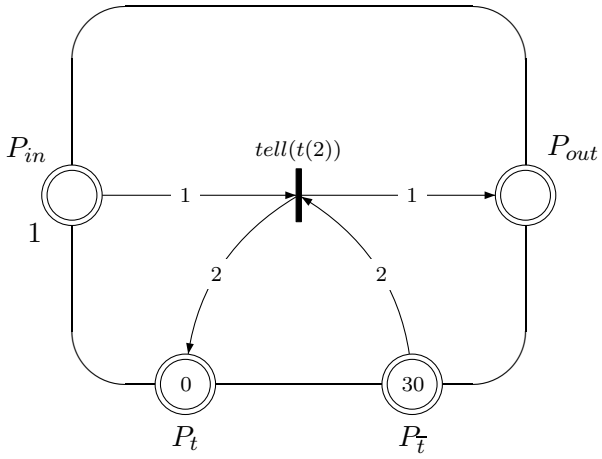


Figure 11.7: The Dense Bach *tell(t(2))* primitive in an Open Petri Net with MAX = 30

Note that, following our developments on page 76, we denote by $\{P_t(m)\}$ the multiset composed of m occurrences of P_t and similarly we denote by $\{P_{\overline{t}}(m)\}$ the multiset composed of m occurrences of $P_{\overline{t}}$.

11.3.1.2 The ask primitive

Figure 11.8 depicts the representation of the *ask* primitive in a DB-open Petri net. As for the *tell* primitive the presence of a dense token *t(m)* of density m on the store is represented by a pair of two internal places: a place P_t and its corresponding anti-place $P_{\overline{t}}$. The place P_t contains m Petri tokens, and the anti-place $P_{\overline{t}}$ contains the difference between the maximum of Petri tokens and m. The presence of all the tokens in $P_{\overline{t}}$ and the absence of tokens in P_t represents the effective absence of a token in the store. The *ask* primitive introduces no new token on the store, as its main action is to verify the effective presence of a token on it. For this reason, there is no link between the anti place $P_{\overline{t}}$ and the transition, but only a bidirectional link between the transition and the place P_t.

The behaviour of the open net associated with the *ask(t(m))* primitive is then the following: the firing of the transition first consumes a token in the entry open place P_{in}, and m Petri tokens in the internal P_t place to verify the presence of m instances of t on the store. Then it produces a token in the exit open place P_{out} and puts m Petri tokens again in the internal place P_t to restore the initial situation. For this reason the arrow between the transition and the place P_t is bi-directional.

With the agent *ask(t(m))* we associate thus the following DB-open Petri Net $(S, T, {}^\bullet(.), (.)^\bullet, O, P_{in}, P_{out}, Pts, M, N)$ on $\mathfrak{T} = \{t\}$:

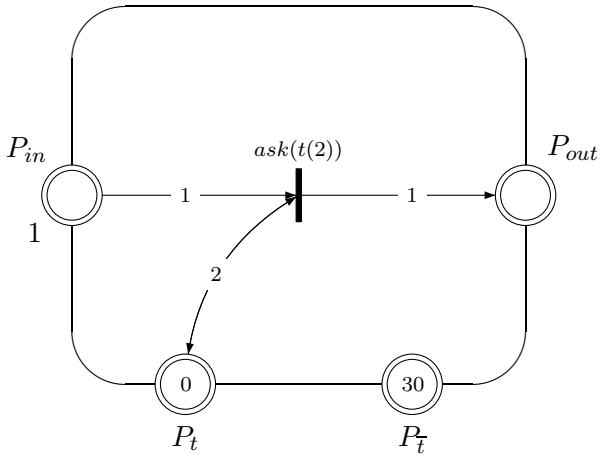


Figure 11.8: The Dense Bach *ask(t(2))* primitive in an Open Petri Net with MAX = 30

- $S = O = \{ P_{in}, P_{out}, P_t, P_{\overline{t}}\}$
- $P_{in} \in O$ is the entry place
- $P_{out} \in O$ is the exit place
- $Pts = \{P_t\} \cup \{P_{\overline{t}}\}$
- $T = \{ask(t(m)) \}$
- ${}^\bullet(ask(t(m))) = \{P_t(m), P_{in}\}$
- $(ask(t(m)))^\bullet = \{P_t(m), P_{out}\}$
- $M(P_t) = 0$ and $M(P_{\overline{t}}) = MAX$
- N = 1

Here we denote again by $\{P_t(m)\}$ the multiset composed of m occurrences of P_t.

11.3.1.3 The get primitive

Figure 11.9 depicts the representation of the *get(t(m))* primitive in a DB-open Petri net. The difference between the *get* and the *ask* primitive is that *get* not only checks for the presence of m occurrences of the token t on the store, but also retrieves them effectively from the store. Both place P_t and anti place $P_{\overline{t}}$ are concerned, but in an opposite way to the *tell* primitive. Indeed, now m instances of the Dense Bach token t must disappear from the store. This is performed by a transfer of m Petri tokens from the place P_t to the anti-place $P_{\overline{t}}$.

The behaviour of the open net associated with the get primitive is the following : the firing of the transition first consumes a token in the entry open place P_{in} and m Petri tokens in the internal P_t to verify the presence of m instances. Then it produces a token in the exit open place P_{out} and puts m Petri tokens in the internal anti-place $P_{\overline{t}}$ to express the decrease of m Dense Bach tokens on the store.

With the agent *get(t(m))* we thus associate the following DB-open Petri Net $(S, T, {}^\bullet(.), (.)^\bullet, O, P_{in}, P_{out}, Pts, M, N)$ on $\mathcal{T} = \{t\}$:

- $S = O = \{ P_{in}, P_{out}, P_t, P_{\overline{t}}\}$
- $P_{in} \in O$ is the entry place
- $P_{out} \in O$ is the exit place
- $Pts = \{P_t\} \cup \{P_{\overline{t}}\}$
- $T = \{get(t(m)\ \}$
- ${}^\bullet(get(t(m))) = \{P_t(m), P_{in}\}$
- $(get(t(m)))^\bullet = \{P_{\overline{t}}(m), P_{out}\}$
- $M(P_t) = 0$ and $M(P_{\overline{t}}) = MAX$
- N = 1

Again, we denote by $\{P_t(m)\}$ the multiset composed of m occurrences of P_t and similarly we denote by $\{P_{\overline{t}}(m)\}$ the multiset composed of m occurrences of $P_{\overline{t}}$.

11.3.1.4 The nask primitive

Finally, Figure 11.10 depicts the representation of the *nask* primitive in a DB-open Petri net. The primitive *nask(t(m))* has a dual behaviour to the *ask* primitive, as it now checks for the presence of minimum $Max - m + 1$ Petri tokens in the anti-place associated with the token t. For this reason, the anti place $P_{\overline{t}}$ is the only concerned. The firing of the transition thus first consumes a Petri token in the open place P_{in}, as well as $MAX - m + 1$ Petri tokens in the anti-place $P_{\overline{t}}$. Then it produces a Petri token in the exit open place P_{in} and replaces the previous $Max - m + 1$ previoulsy consumed Petri tokens in the same place anti-place $P_{\overline{t}}$.

With the agent *nask(t(m))* we thus associate the following DB-open Petri Net $(S, T, {}^\bullet(.), (.)^\bullet, O, P_{in}, P_{out}, Pts, M, N)$ on $\mathcal{T} = \{t\}$:

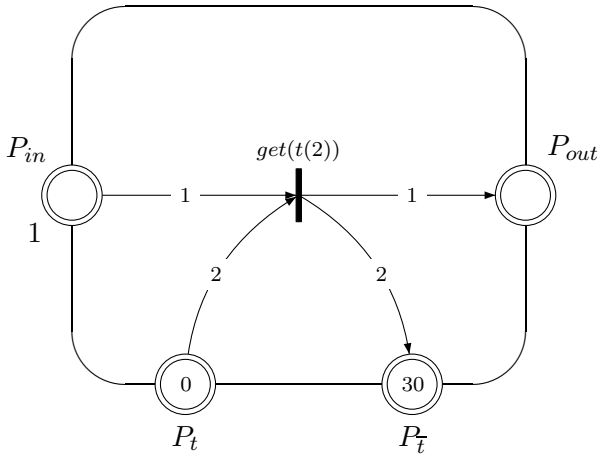


Figure 11.9: The Dense Bach *get(t(2))* primitive in an Open Petri Net with MAX = 30

- $S = O = \{ P_{in}, P_{out}, P_t, P_{\overline{t}}\}$
- $P_{in} \in O$ is the entry place
- $P_{out} \in O$ is the exit place
- $Pts = \{P_t\} \cup \{P_{\overline{t}}\}$
- $T = \{nask(t(m)) \}$
- ${}^\bullet(nask(t(m))) = \{P_{\overline{t}}(MAX - m + 1), P_{in}\}$
- $(nask(t(m)))^\bullet = \{P_{\overline{t}}(MAX - m + 1), P_{out}\}$
- $M(P_t) = 0$ and $M(P_{\overline{t}}) = MAX$
- N = 1

Again we denote by $\{P_{\overline{t}}(m)\}$ the multiset composed of m occurrences of $P_{\overline{t}}$.

11.3.2 The complex agents

After the representation of the basic Dense Bach primitives in a open Petri Net, the next step consists in studying the representation of complex Dense Bach agents. These complex agents result either from the composition of basic primitives, or from the composition of other complex agents. In general, an agent defines what we shall call a *block*. As for the primitives, any block must respect the basic open Petri net structure, as depicted in Figure 11.6. The structure must

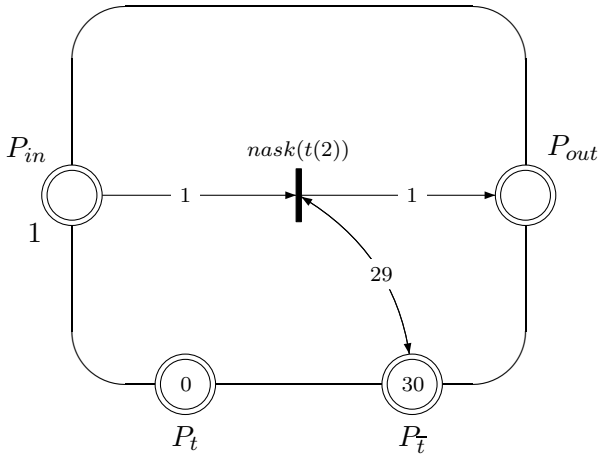


Figure 11.10: The Dense Bach *nask(t(2))* primitive in an Open Petri Net with MAX = 30

start with an open place and finish also with an open place. In general, the composition of different blocks is achieved by merging the exit open place of the first block of the composition with the entry open place of the next block. However some constructions will need specific internal places (i.e not affected by the environment) and transitions for the correct execution of the considered agent.

11.3.2.1 The sequential operator

The first complex structure of two agents results from their sequential composition. It is obtained by merging the open output place of the first agent with the open input place of the second agent. No new transition are added. Only the post-condition for firing the transition leading to the output place of the first agent must be adapted, in the sense that the targeted place must now be the input place of the second agent in the sequential composition, and also its weight is to be adapted to the tokens assumed by the second agent. Regarding the token space, every agent still refers to its internal places and anti-places. Figure 11.12 depicts the construction of the DB-open Petri Net associated with the sequential composition of the two primitives *tell(a(2))* and *tell(b(3))*, whose associated DB-open Petri Nets are represented in Figure 11.11. The places and anti-places of the tokens a and b are represented at the bottom of the figure. With no token in the places in black, and a maximum of 30 tokens in the anti-places in red, this picture represents the initial situation of this sequential agent, starting from an empty store.

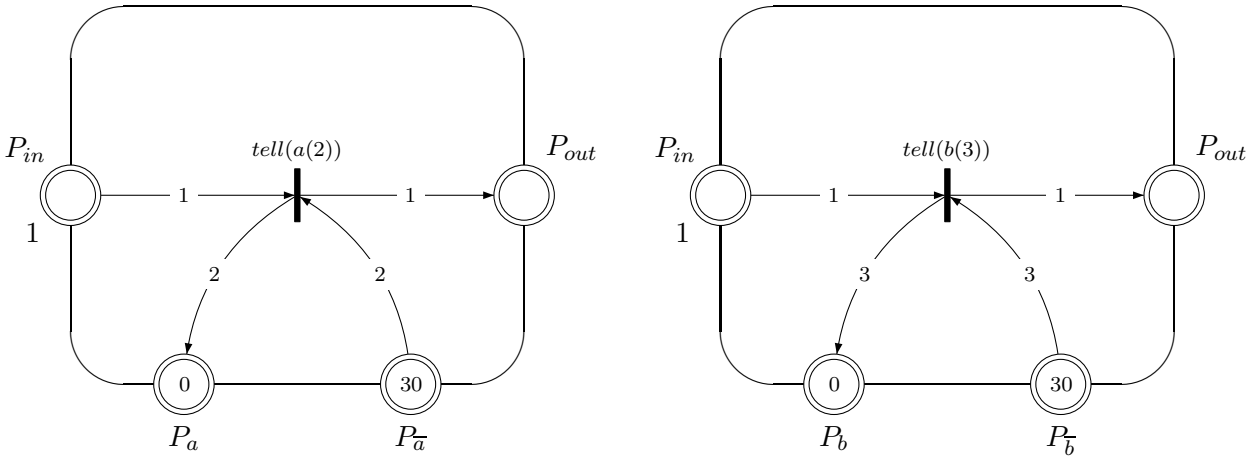


Figure 11.11: The two Dense Bach *tell(a(2))* and *tell(b(3))* primitives to be combined sequentially

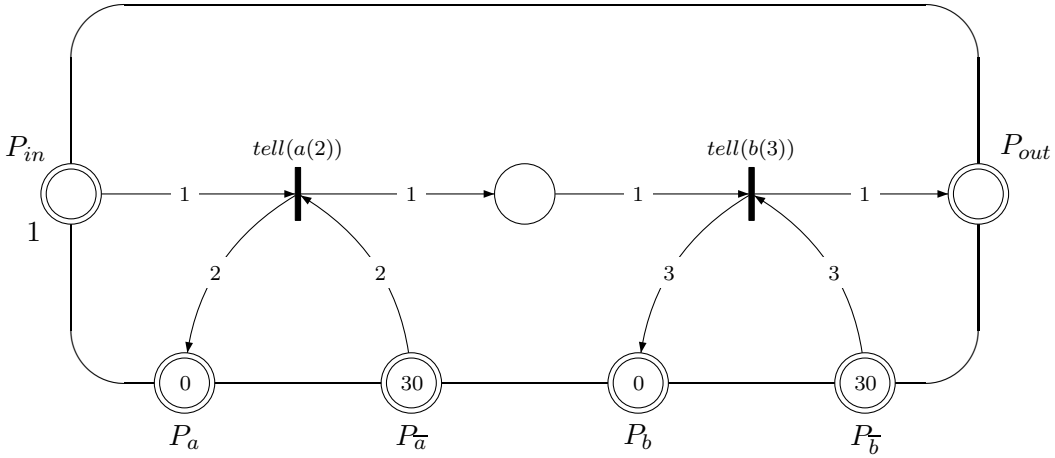


Figure 11.12: The effective sequential composition of *tell(a(2))* and *tell(b(3))*

Let us now look how the construction of a sequential composition proceeds with two agents expressed in a generic form. Figure 11.13 represents the two generic agents in their general form. For the ease of representation, only two transitions connected to the entry places P_{in_1} and P_{in_2} and two connected to the exit places P_{Out_1} and P_{Out_2} have been represented. The tokens on the store, manipulated by the primitives, are respectively represented by the sets $\{t_{1,1}, \ldots, t_{1,I}\}$ and $\{t_{2,1}, \ldots, t_{2,J}\}$. Finally the number of tokens in P_{in_1} that are necessary for the execution of the first agent is equal to N_1, and in place P_{in_2} of the second agent is equal to N_2.

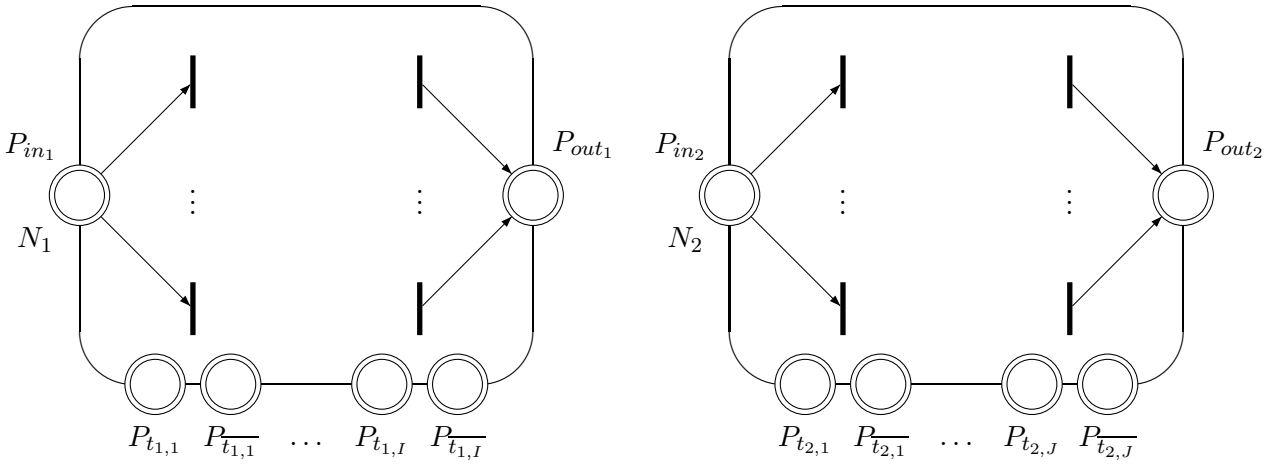


Figure 11.13: The two generic agents to be combined sequentially

Using the notations of Definition 29, with the first agent A_1 we associate the DB-open Petri Net $(S_1, T_1, {}^\bullet(.)_1, (.)_1^\bullet, O_1, P_{in_1}, P_{out_1}, Pts_1, M_1, N_1)$ on $\mathfrak{T}_1$, defined as follows:

- $\mathfrak{T}_1$: the set of tokens of the considered agent i.e. $\{t_{1,1}, \ldots, t_{1,I}\}$
- S_1 : its places
- P_{in_1} : entry place
- P_{out_1} : exit place
- $Pts_1 = \{P_{t_{1,1}}, P_{\overline{t_{1,1}}}, \ldots, P_{t_{1,I}}, P_{\overline{t_{1,I}}}\}$
- $O_1 = Pts_1 \cup \{P_{in_1}, P_{out_1}\} \subseteq S_1$
- T_1 : its transitions
- ${}^\bullet(.)_1$ and $(.)_1^\bullet : T_1 \to S^\oplus$
- $M_1 : S_1 \to \mathbb{N}$ is a partial marking such that:
  - $M(P_{t_{1,i}}) = 0$ for any $P_{t_{1,i}} \in Pts_1$
  - $M(P_{\overline{t_{1,i}}}) = MAX$ for any $P_{\overline{t_{1,i}}} \in Pts_1$
- N_1 is a strictly positive number

Similarly, with the second agent A_2 we associate the DB-open Petri Net $(S_2, T_2, {}^\bullet(.)_2, (.)_2^\bullet, O_2, P_{in_2}, P_{out_2}, Pts_2, M_2, N_2)$ on $\mathfrak{T}_2$, defined as follows:

- $\mathfrak{T}_2$: the set of tokens of the considered agent i.e. $\{t_{2,1}, \ldots, t_{2,J}\}$
- S_2 : its places
- P_{in_2} : entry place
- P_{out_2} : exit place
- $Pts_2 = \{P_{t_{2,1}}, P_{\overline{t_{2,1}}}, \ldots, P_{t_{2,J}}, P_{\overline{t_{2,J}}}\}$
- $O_2 = Pts_2 \cup \{P_{in_2}, P_{out_2}\} \subseteq S_2$
- T_2 : its transitions
- ${}^\bullet(.)_2$ and $(.)_2^\bullet : T_2 \to S^\oplus$
- $M_2 : S_2 \to \mathbb{N}$ is a partial marking such that:
  - $M(P_{t_{2,i}}) = 0$ for any $P_{t_{2,i}} \in Pts_2$
  - $M(P_{\overline{t_{2,i}}}) = MAX$ for any $P_{\overline{t_{2,i}}} \in Pts_2$
- N_2 is a strictly positive number

The first step of the construction consists in inserting the two previous agents in a block representing their sequential composition. This block has a DB-open Petri Net form. It has one entry place P_{in} and one exit place P_{out}. The DB-open Petri Net representations of the two agents to be composed are represented in dashed form. This signals that their components will be modified by the construction process. Figure 11.14 represents the sequential agent including the two Open Petri net agents.

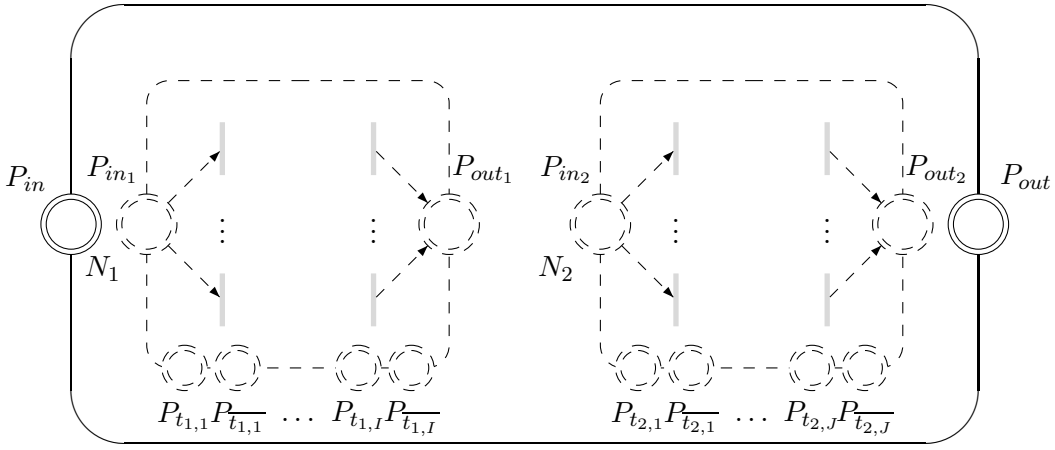


Figure 11.14: The two agents inside the sequential composition

The second step of the construction consists in making the places and anti-places associated with the tokens $\{P_{t_{1,1}}, \ldots, P_{t_{1,I}}, P_{t_{2,1}}, \ldots, P_{t_{2,J}}\}$ visible to the environment of the sequential composition. This is done by translating the respective places and anti-places from the borders of the Open Petri net representation of the agents to the border of the sequential Open Petri net. It is obvious that the tokens shared by both agents will appear only once on the border of the sequential agent. Figure 11.15 represents the externalization of the places and anti-places associated with the tokens.

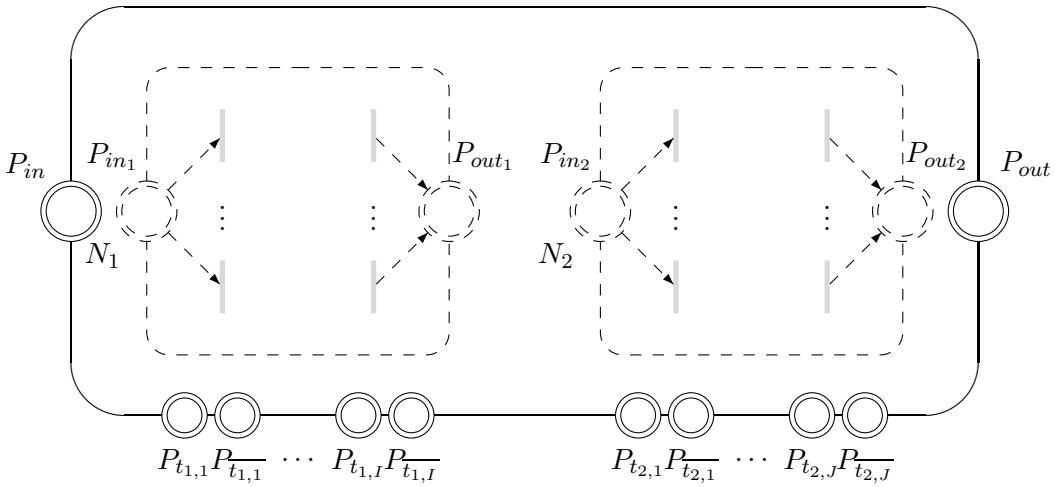


Figure 11.15: The places of the tokens visible outside

The third step consists, on the one hand, in a fusion between the entry place P_{in_1} of the first agent and the entry place P_{in} of the sequential agent and, on the other hand, in a fu-

sion bewteen the exit place P_{out_2} of the second agent and the exit place P_{Out} of the sequential agent. This implies that the number N_1 of tokens necessary for the execution of the agent is now required by the entry place of the sequential agent. Figure 11.16 represents that transformation.

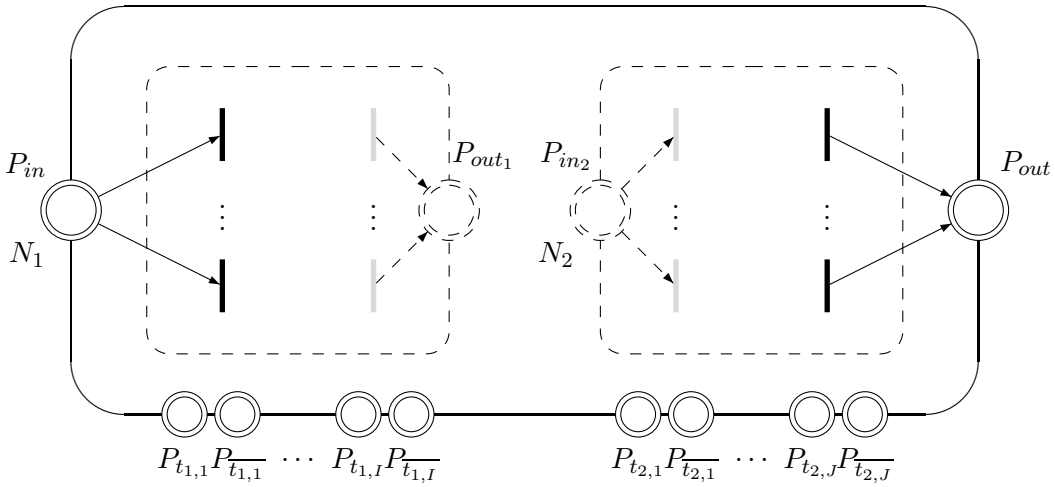


Figure 11.16: The fusion of the entry places P_{in} with P_{in_1} and from P_{out} with P_{out_2}

The fourth and final step of the construction consists in grouping the exit place P_{out_1} of the first agent with the entry place P_{in_2} of the second agent. This implies to adpat the number of tokens in the exit place P_{out_1}, i.e 1, to the number N_2 required by the second agent for its execution. All other places and transitions are preserved with their marking and weight. Figure 11.17 represents that last transformation.

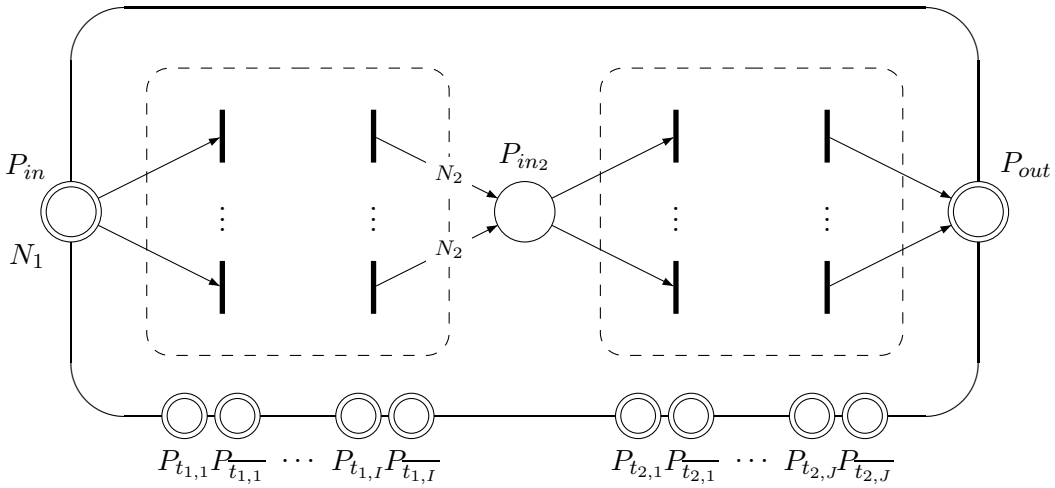


Figure 11.17: The fusion of P_{out_1} and P_{in_2}

Again in the notations of Definition 29, with the sequential composition $A_1 \; ; \; A_2$ we associate the following DB-open Petri Net $(S, T, {}^\bullet(.), (.)^\bullet, O, P_{in}, P_{out}, Pts, M, N)$ on $\mathcal{T}$:

- $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2$
- $P_{in} = P_{in_1}$ is its entry place
- $P_{out} = P_{out_2}$ is its exit place
- $Pts = Pts_1 \cup Pts_2$
- $O = \{P_{in}, P_{out}\} \cup Pts$
- $S = (S_1 \backslash O_1) \cup (S_2 \backslash O_2) \cup O \cup \{P_{in_2}\}$
- $M = \begin{cases} M_1(P) & if \quad P \in S_1 \backslash O_1 \\ M_2(P) & if \quad P \in S_2 \backslash O_2 \\ 0 & if \quad \exists t : P = P_t \\ MAX & if \quad \exists t : P = P_{\overline{t}} \end{cases}$
- $T = T_1 \cup T_2$
- ${}^\bullet(T) = \begin{cases} {}^\bullet(T)_1 & if \quad T \in T_1 \\ {}^\bullet(T)_2 & if \quad T \in T_2 \end{cases}$
- $(T)^\bullet = \begin{cases} (T)^\bullet_1 & where \quad P_{out_1} \quad is\ replaced\ by \quad P_{in_2}(N_2) \\ (T)^\bullet_2 & if \quad T \in T_2 \end{cases}$

It appears that after the construction, the set S is obtained as the union of the sets S_1 and S_2, without their sets of open places O_1 and O_2, with the set O and with the set containing the only place P_{in_2}. Morevover, the set O of open places is equal to the union of $\{P_{in}, P_{out}\}$ with the set Pts of the places and anti-places associated with the tokens. The number of tokens in the entry place must be equal to N_1, and the number of tokens required in the internal place P_{in_2} is equal to N_2.

11.3.2.2 The parallel operator

Figure 11.19 depicts the construction of the parallel composition of two agents, described in Figure 11.18, in a DB-open Petri Net. In a parallel composition, every agent participating to the composition must be succesful, in order to produce a global succesful result. This implies that the entry open place P_{in} must contain the number of tokens necessary to activate every sub-agent participating to the parallel composition. This is obtained by summing the number of tokens

needed by these sub-agents to be executed. In Figure 11.19, related to $tell(a(2)) \parallel tell(b(2))$, the required number of tokens in P_{in} is equal to 2. Moreover during the execution of the composition, every agent must be selected only once. In order to guarantee this condition, some interblocking mechanisms must be introduced. This is done by associating an internal place with the Petri net representation of every sub-agent, as part of the pre-condition for firing its transition. In our figure, these places are respectively named P_{aux_1} and P_{aux_2}, and are initially marked with 1. In accordance to our model, the entry place P_{in} merges both entry places of the primitives *tell(a(2))* and *tell(b(2))*. Regarding the output place of the parallel agent, it does not result from the direct merging of both output places of the *tell(a(2))* and *tell(b(2))* primitives. Indeed the output place of the parallel agent is reached only when both primitives have been sucessfully executed. Moreover to allow a possible iteration the action of supplying the final output place of the open Petri net must be done together with a restoring of the initial state of both control places P_{aux_1} and P_{aux_2}. For these reasons, an internal transition T is added to the structure of the parallel agent. This transition collects both successful results of the primitives *tell(a(2))* and *tell(b(2))*, and provides respectively 1 token to the output and control places. The restoring of the control places to their initial state means that the structure is able to manage iterative agent. We have implemented the restoring, although we do not consider in our thesis the case of iterative Dense Bach agents. Again the figure represents the initial situation of the complex agent starting from an empty store. The places and anti-places of tokens a and b are represented on the border of the DB-open Petri Net for the parallel composition, with a value MAX equals to 30.

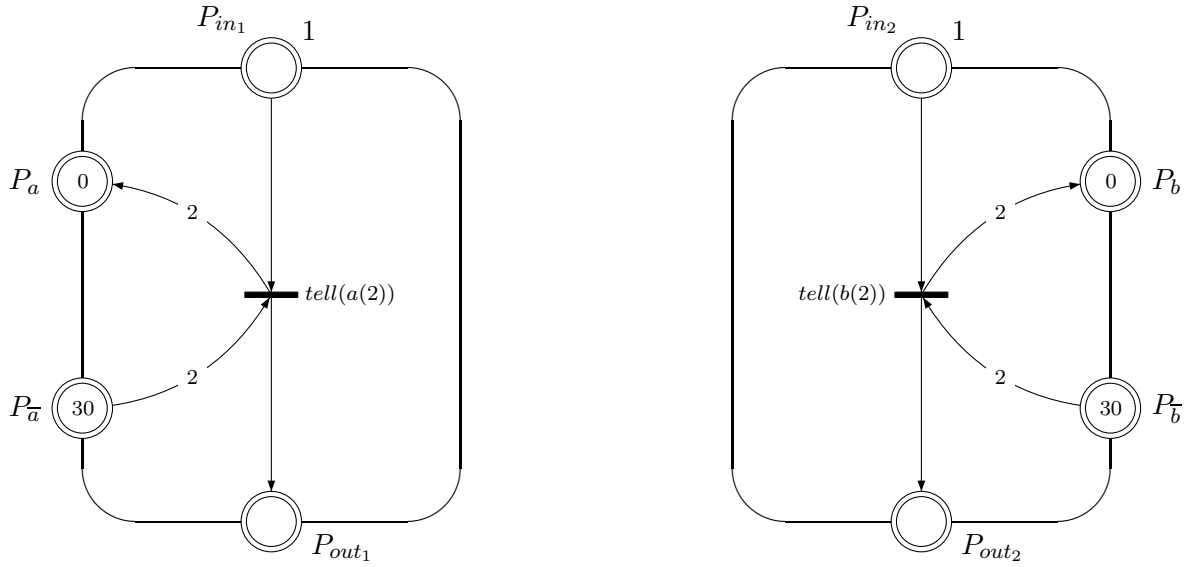


Figure 11.18: The two Dense Bach *tell(a(2))* and *tell(b(2))* primitives to be composed in parallel

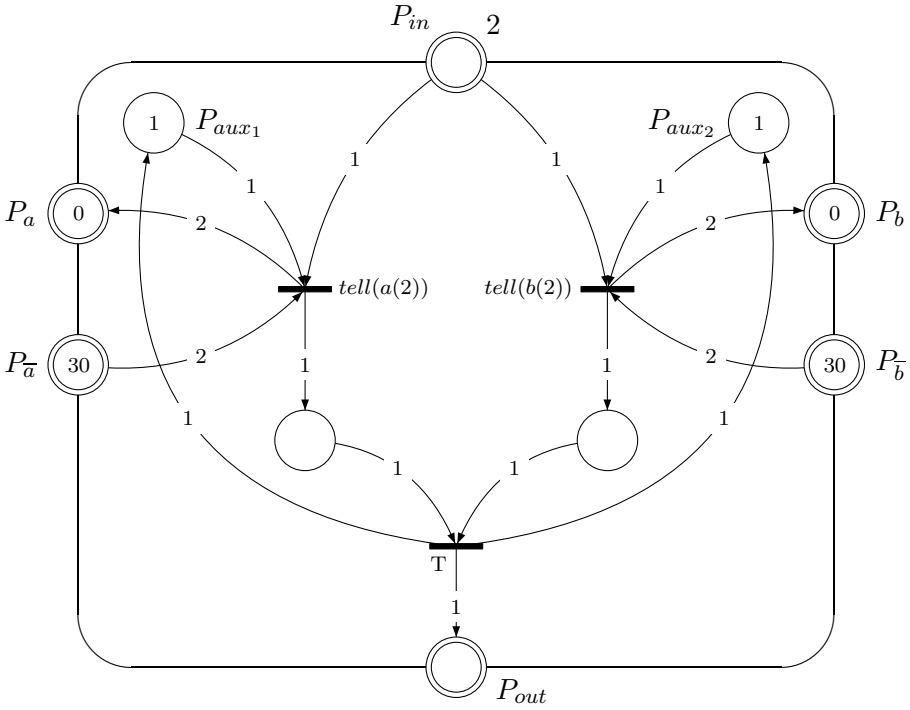


Figure 11.19: The effective parallel composition of *tell(a(2))* and *tell(b(2))*

Figure 11.20 represents the two agents in a DB-open Petri Net general form, that must be combined in parallel. For the ease of the drawing of the picture, the representation of both agents have been rotated from 90 degrees on the right, with their entry places on the top of the drawing.

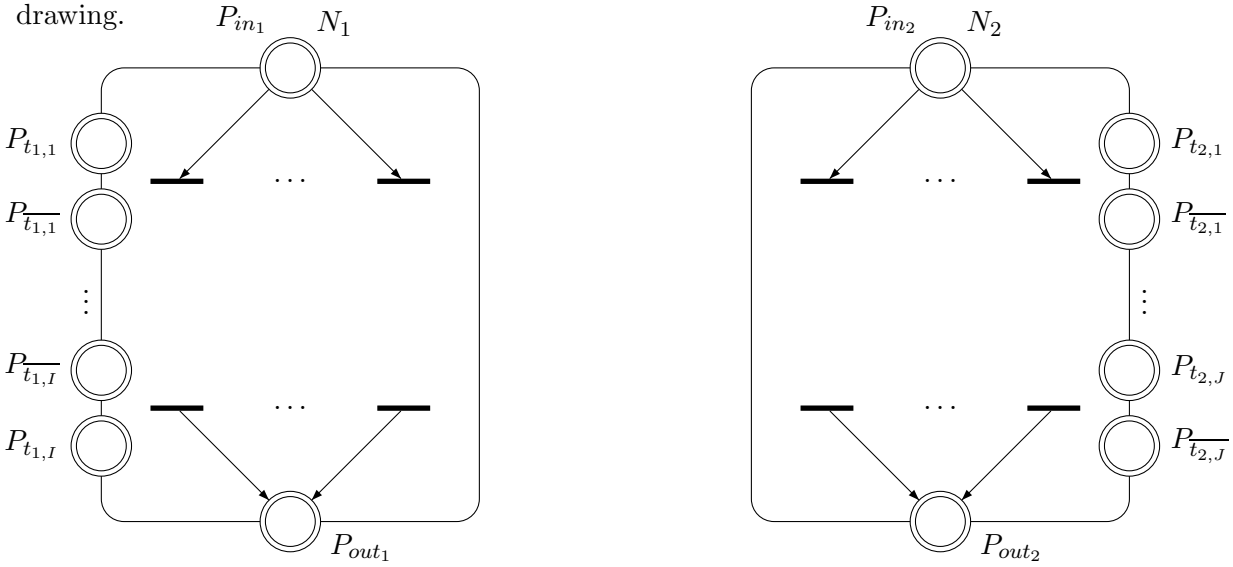


Figure 11.20: The two Dense Bach agents to be composed in parallel

As for the sequential composition these two generic agents A_1 and A_2 are associated respectively with two DB-open Petri Net $(S_1, T_1, {}^\bullet(.)_1, (.)_1^\bullet, O_1, P_{in_1}, P_{out_1}, Pts_1, M_1, N_1)$ on $\mathfrak{T}_1$ and

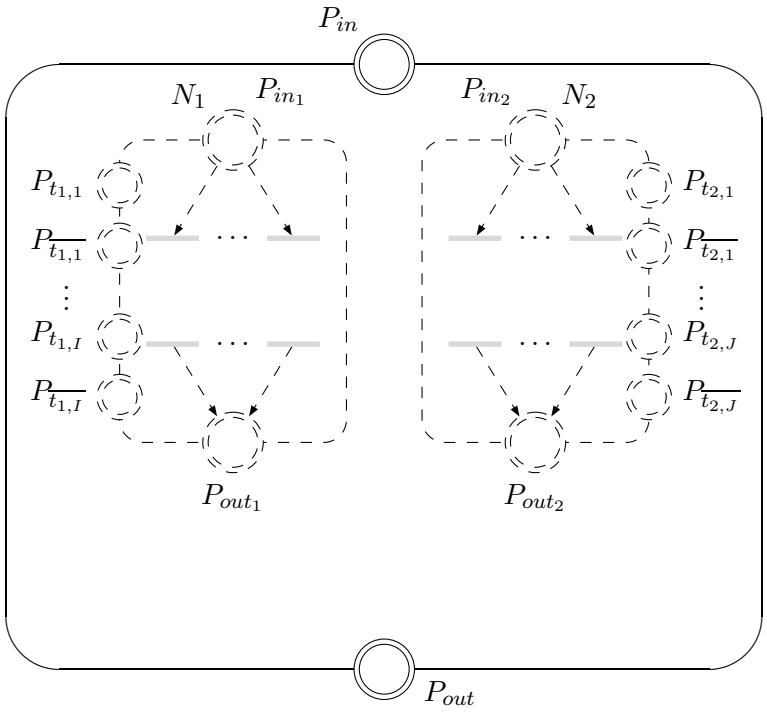


Figure 11.21: The two agents in the parallel composition

$(S_2, T_2, {}^\bullet(.)_2, (.)^\bullet_2, O_2, P_{in_2}, P_{out_2}, Pts_2, M_2, N_2)$ on $\mathfrak{T}_2$. They are defined exactly as in page 301.

The first step in the construction of the parallel composition consists in inserting the two agents to be combined inside the DB-open Petri Net representation of the parallel agent. This representation has only two places : an entry and an exit. Figure 11.21 represents this insertion.

After the insertion, the places and anti-places $\{P_{t_{1,1}}, \ldots, P_{t_{1,I}}, P_{t_{2,1}}, \ldots, P_{t_{2,J}}\}$ migrate on the border of the parallel agent. They become open places visible from the outside environment of the parallel agent. Figure 11.22 represents this transformation.

The next step represents the first transformation of the agents, where some of their elements become part of the element of the parallel agent. It concerns the entry places P_{in_1} and P_{in_2} that will merge with the entry place P_{in}. For a parallel composition, all subagents must be executed. This implies that if the respective number of tokens of the two subagents necessary for their execution are N_1 and N_2, the resulting number for the parallel agent must be equal to their summation, i.e. $N_1 + N_2$. Starting from the new entry place P_{in} an arrow must point on every transition that is part of the subagents. Figure 11.23 show that step of the transformation.

Symmetrically, after the connection of the subagents with the new entry place comes also

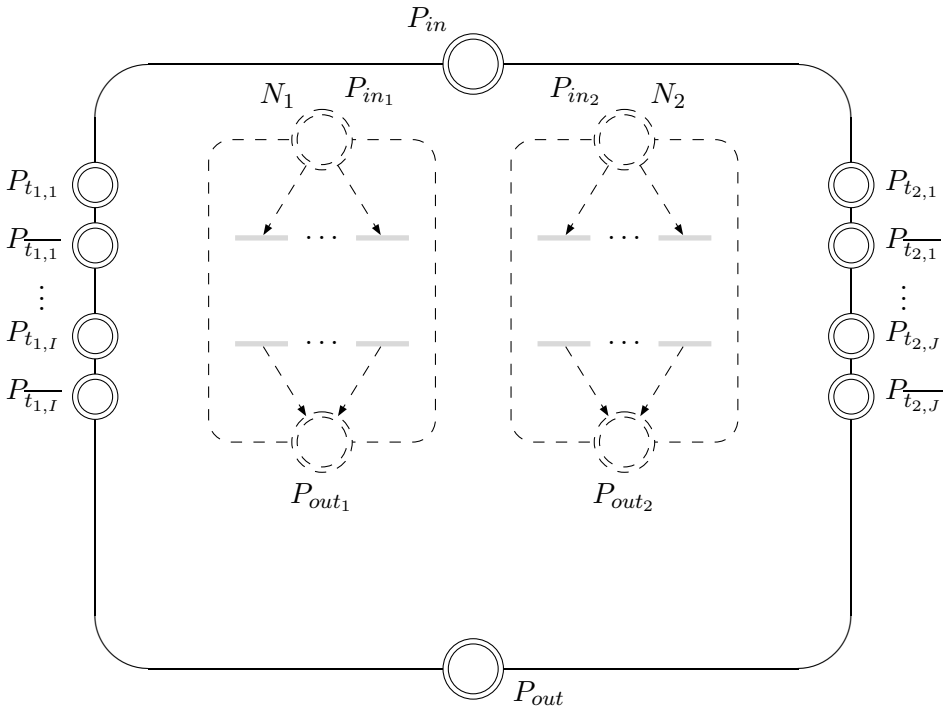


Figure 11.22: Tokens are visible outside

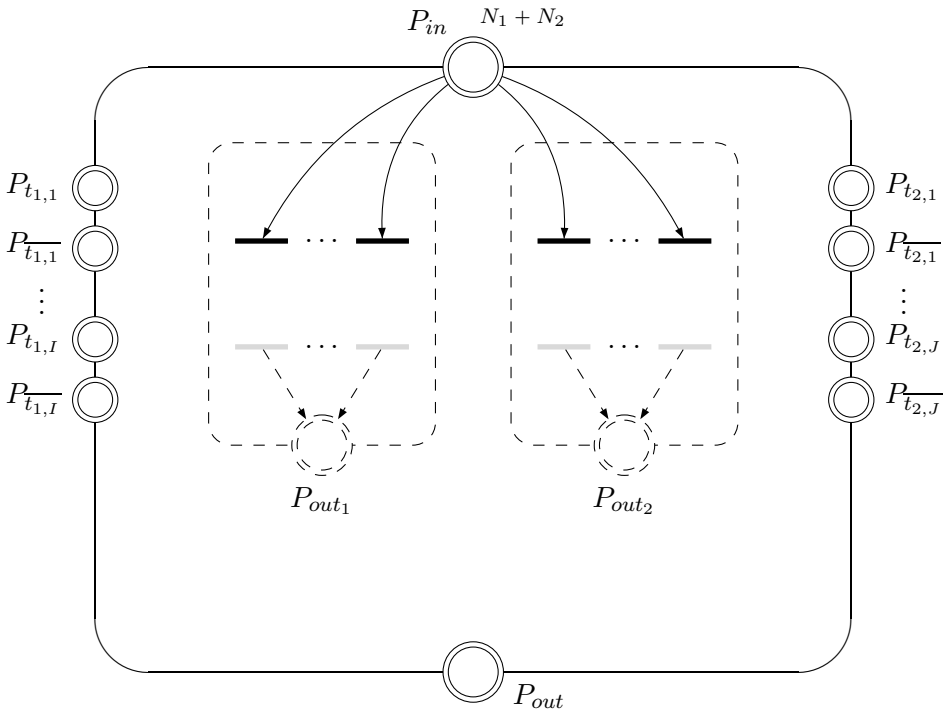


Figure 11.23: Entry place of parallel agent connected to transitions

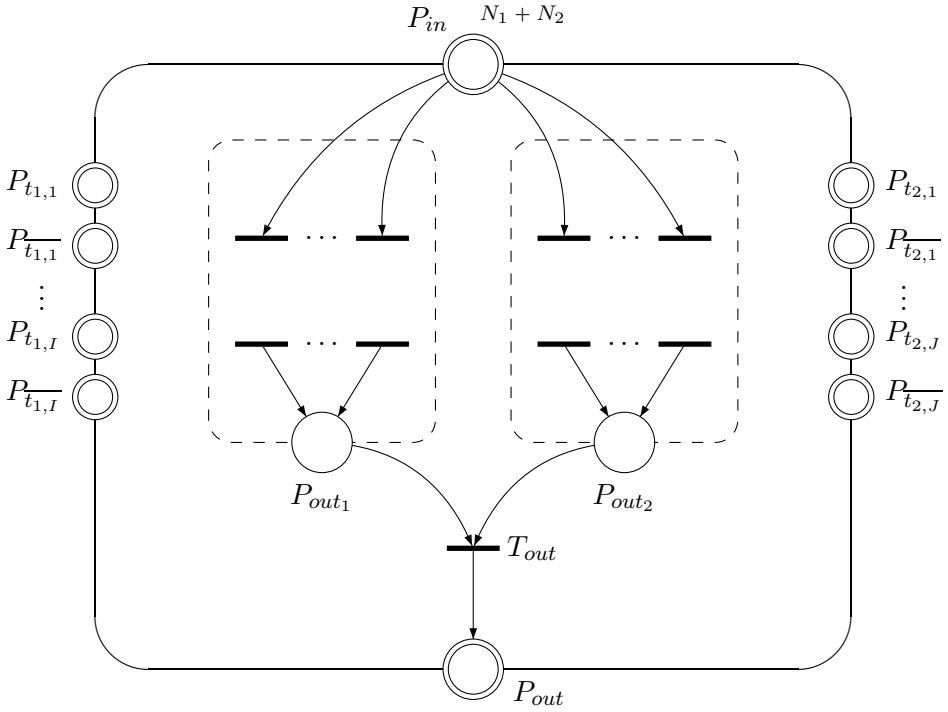


Figure 11.24: Connection with exit place of parallel agent

the necessity to connect the outpout places P_{out_1} and P_{out_2} of the subagents, with the output place P_{out} of the parallel agent. This operation requires the introduction of a transition T_{out}. The places P_{out_1} and P_{out_2} become standard places (i.e. non open) of the Net. Figure 11.24 depicts its connection with the output place P_{out}.

As explained in the introduction, as every subagent must be executed, it is necessary to introduce a mechanism that ensures that a subagent is executed only one time. Indeed, as the number of the tokens present inside the entry place P_{in} is equal to $N_1 + N_2$, it is possible for a same transition inside a subagent to be selected more than once. To avoid such incorrect behaviour, we introduce internal auxiliary places P_{aux_1} and P_{aux_2}. These places are not visible from outside the parallel agent and are only used for an internal regulation of the parallel agent. Each of these places is associated with a subagent. Figure 11.25 shows the introduction of these two places, P_{aux_1} near the first subagent, and P_{aux_2} near the second subagent.

For a specific subagent, its associated auxiliary place will take care of the firing of every transition only once. This implies for an auxiliary place to be initialized with a number of tokens required by the subagent. In our schema, this number is equal to N_1 for the first subagent, and is equal to N_2 for the second subagent. Figure 11.26 shows the connection between the auxiliary places and each transitions of their respective associated subagent.

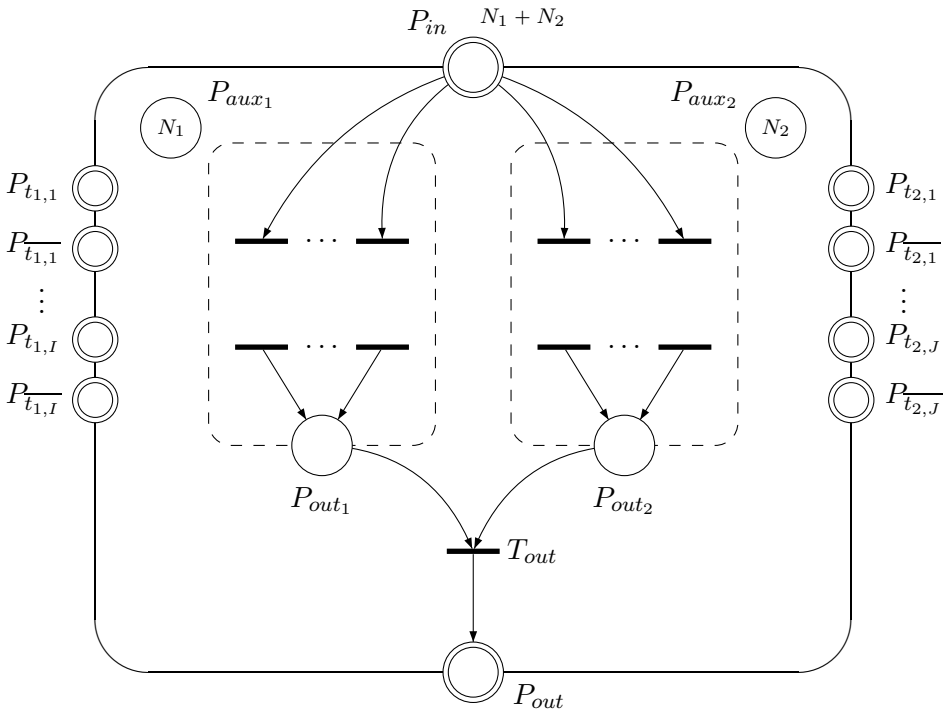


Figure 11.25: Introduction of auxiliary places

At the end of the execution, when the internal transition T_{out} is fired after the successful execution of all the subagents, the state of the auxiliary places is restored to their initial values, i.e. respectively N_1 and N_2. Figure 11.27 shows that last step of the execution.

Again with the notations of Definition 29, the resulting DB-open Petri Net of the parallel composition $A_1 \parallel A_2$ is the DB-open Petri Net $(S, T, {}^\bullet(.), (.)^\bullet, O, P_{in}, P_{out}, Pts, M, N)$ on $\mathcal{T}$ defined as follows:

- $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2$
- P_{in} is its (fresh) entry place
- P_{out} is its (fresh) exit place
- Pts $= Pts_1 \cup Pts_2$
- $O = \{P_{in}, P_{out}\} \cup Pts$
- $S = (S_1 \backslash O_1) \cup (S_2 \backslash O_2) \cup O \cup \{P_{out_1}, P_{out_2}, P_{aux_1}, P_{aux_2}\}$
- $T = T_1 \cup T_2 \cup \{T_{out}\}$

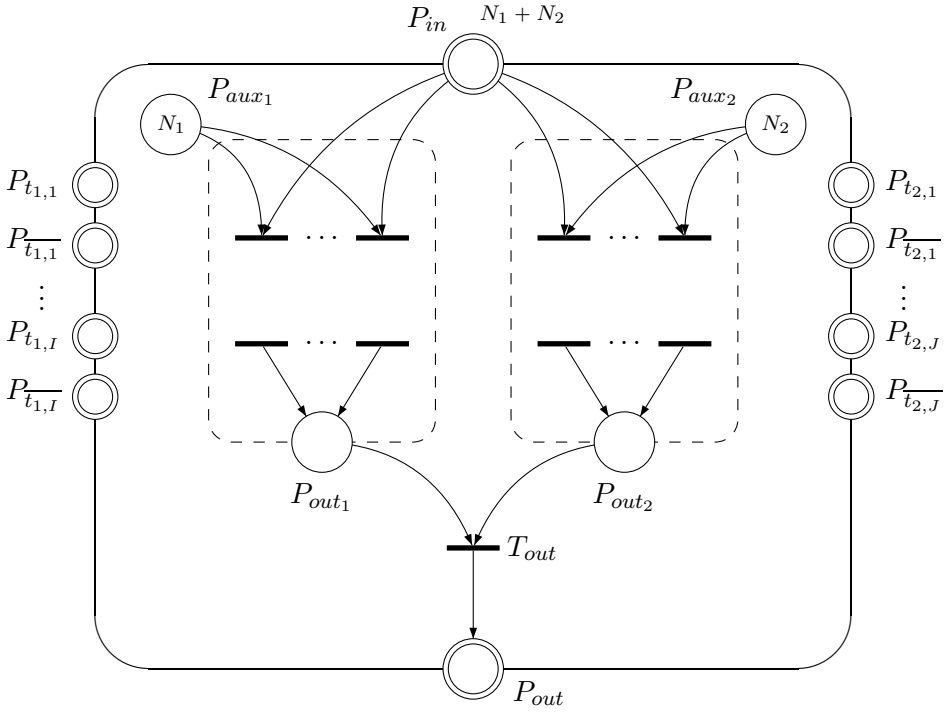


Figure 11.26: Introduction of auxiliary places (cont)

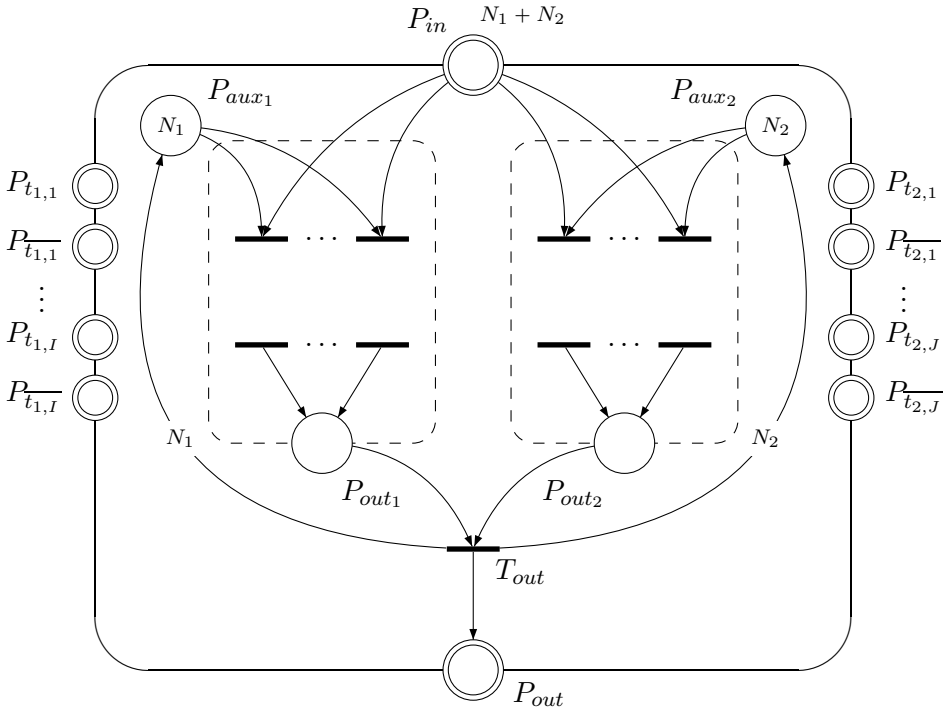


Figure 11.27: Introduction of auxiliary places (final)

- $^\bullet(T) = \begin{cases} {}^\bullet(T)_1 & where \quad \{P_{in_1}\} \quad is\ replaced\ by \quad \{P_{in}, P_{aux_1}\} \quad if \quad T \in T_1 \\ {}^\bullet(T)_2 & where \quad \{P_{in_2}\} \quad is\ replaced\ by \quad \{P_{in}, P_{aux_2}\} \quad if \quad T \in T_2 \end{cases}$
- $^\bullet(T_{out}) = \{P_{out_1}, P_{out_2}\}$
- $(T)^\bullet = \begin{cases} (T)^\bullet_1 & if \quad T \in T_1 \\ (T)^\bullet_2 & if \quad T \in T_2 \end{cases}$
- $(T_{out})^\bullet = \{P_{out}, P_{aux_1}(N_1), P_{aux_2}(N_2)\}$
- $M(P_t) = 0 \quad \forall t \in \mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2$
- $M(P_{\overline{t}}) = MAX \quad \forall \overline{t} \in \mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2$
- $M(P_{aux_1}) = N_1$
- $M(P_{aux_2}) = N_2$
- $M(P) = \begin{cases} M_1(P) & \text{for any } P \in S_1 \backslash O_1 \\ M_2(P) & \text{for any } P \in S_2 \backslash O_2 \end{cases}$
- $N = N_1 + N_2$

11.3.2.3 The choice operator

We proceed now with the construction of the choice composition of two agents in a DB-open Petri net structure as we have done with the parallel operator. However this model must be constructed carefully, in order to avoid the production of some incorrect results. In particular, the choice between two agents means that the execution of one of them excludes the execution of the second. The Petri net structure must reflect this constraint. As for the parallel composition, the construction introduces some interblocking mechanisms, in the form of two auxiliary control places P_{aux_1} and P_{aux_2}. Figure 11.29 shows the complete open net corresponding to the choice between the two agents of Figure 11.28. To be fired, a transition associated with a sub-agent not only consumes the token present in its own control place, but also checks i.e. consumes and replaces the token present in the control place associated with the other transition. This permits to check if the other agent of the choice composition has not yet started its execution. An absence of tokens in a control place implies then, on the one hand, that its associated transition has been already fired, and, on the other hand, that no other transition can now be fired.

Moreover as for the parallel case, the construction of the choice respects our model of Figure 11.6. This means that both entry places of the sub-agents are merged inside the open entry place P_{in} of the choice agent. On the other hand, for the open output place P_{out}, the exclusive

execution of one sub-agent implies the presence of two internal transitions T_{out_1} and T_{out_2}, with the same role to provide the token in the open output place P_{out} and to restore the state of its correspondant control place.

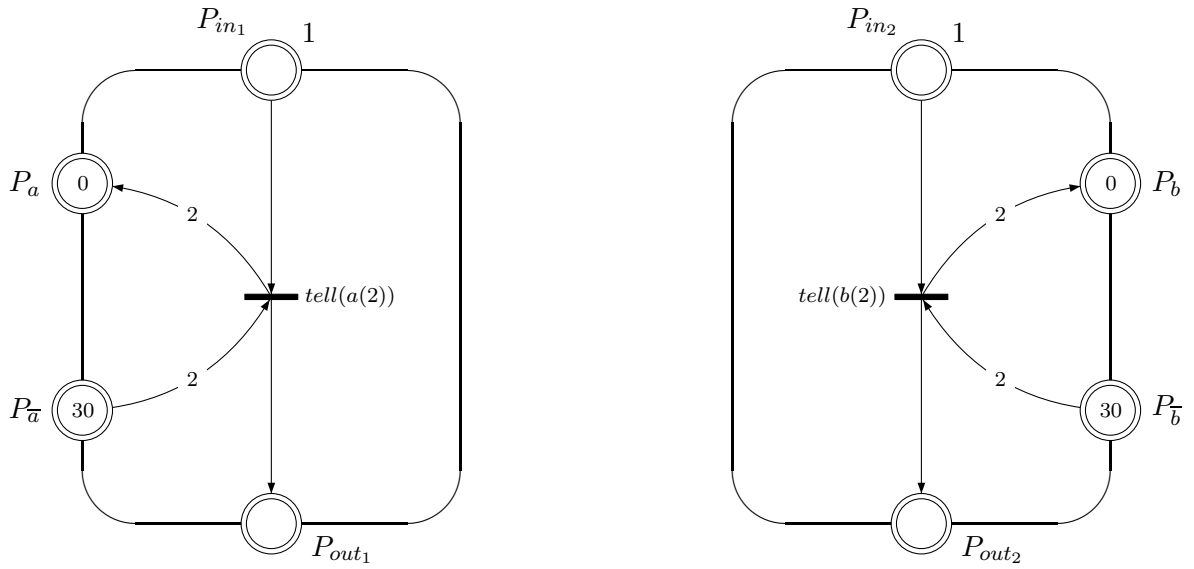


Figure 11.28: The two Dense Bach *tell(a(2))* and *tell(b(2))* primitives to be composed in a choice

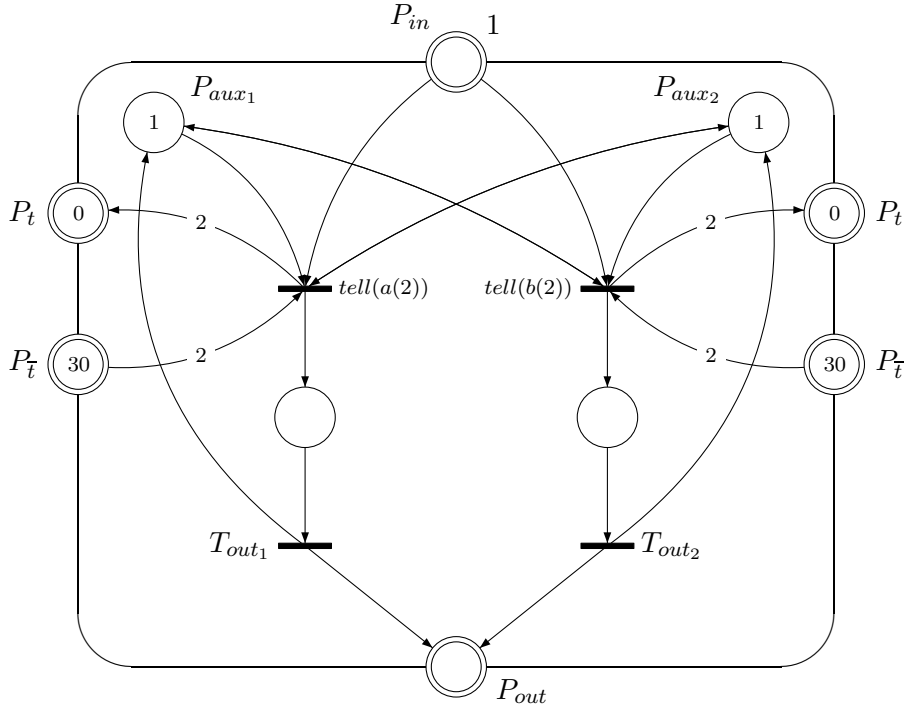


Figure 11.29: The effective choice composition of *tell(a(2))* and *tell(b(2))*

Figure 11.30 shows the representation, as DB-open Petri Net, of the two generic subagents that must be included inside a choice compositional agent. Again, for the ease of the drawing of

the picture, the representation of both agents have been rotated from 90 degrees on the right, with their entry places on the top of the drawing. The number of tokens required by every subagent to be successfully executed is N_1 for the first one and N_2 for the second one. We will suppose for the rest of the discussion that $N_1 \geqslant N_2$.

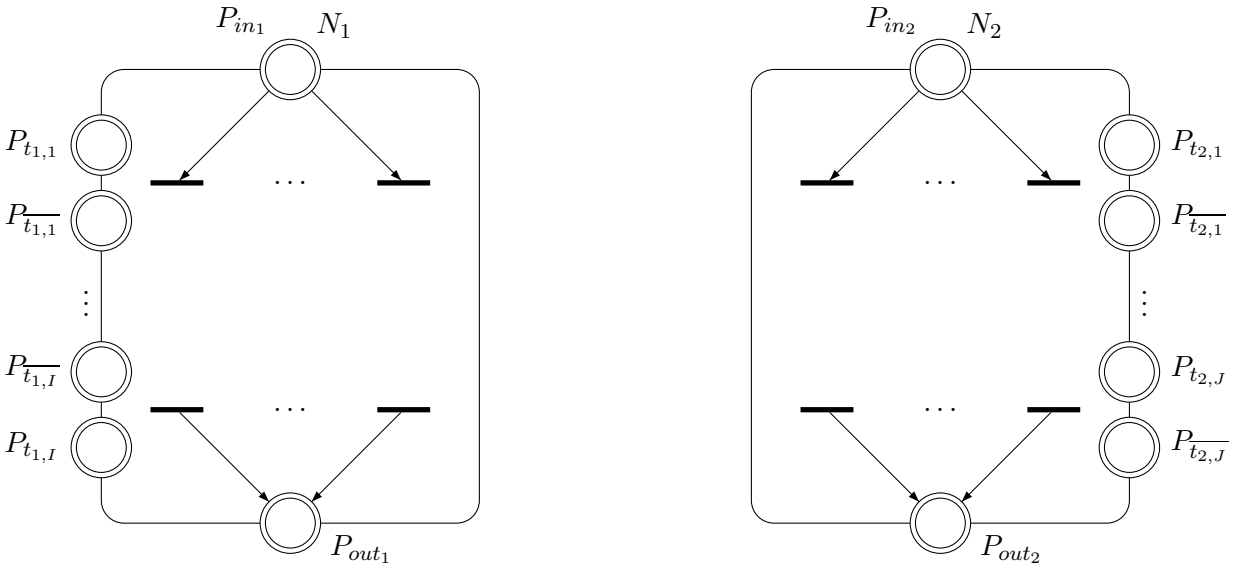


Figure 11.30: The two Dense Bach agents to be composed in a choice

As for the sequential and the parallel compositions these two generic agents A_1 and A_2 are associated respectively with two DB-open Petri Nets $(S_1, T_1, {}^\bullet(.)_1, (.)_1^\bullet, O_1, P_{in_1}, P_{out_1}, Pts_1, M_1, N_1)$ on $\mathfrak{T}_1$ and $(S_2, T_2, {}^\bullet(.)_2, (.)_2^\bullet, O_2, P_{in_2}, P_{out_2}, Pts_2, M_2, N_2)$ on $\mathfrak{T}_2$. They are defined exactly as in page 301.

As for the parallel composition, the first step of the construction consists in inserting the two subagents inside the structure of the parallel agent. Figure 11.31 represents that operation.

The places and anti-places of $\{P_{t_{1,1}}, \ldots, P_{t_{1,I}}, P_{t_{2,1}}, \ldots, P_{t_{2,J}}\}$ staying visible to the outside environment, they migrate to the border of the parallel agent, as Figure 11.32 shows it.

The entry place P_{in} of the parallel agent is substituted to the entry places P_{in_1} and P_{in_2} of the two subagents. In consequence all the transitions of the two subagents connected to their respective entry place P_{in_1} and P_{in_2} are now connected to the only entry place P_{in}. Regarding the number of tokens in the common entry place, it must reflect the fact that only one of the two subagent has to be executed. This implies that the number of tokens necessary for the

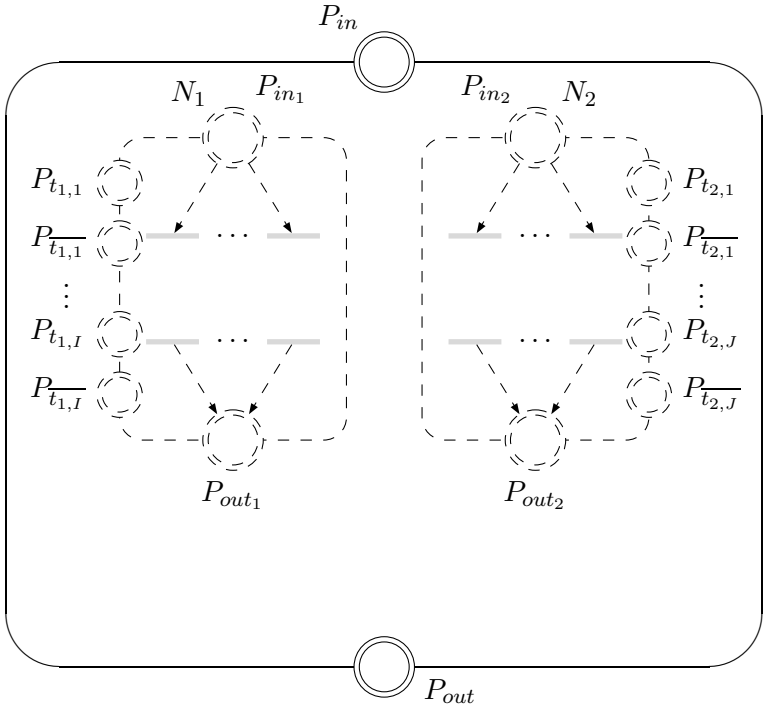


Figure 11.31: The two agents in the choice composition

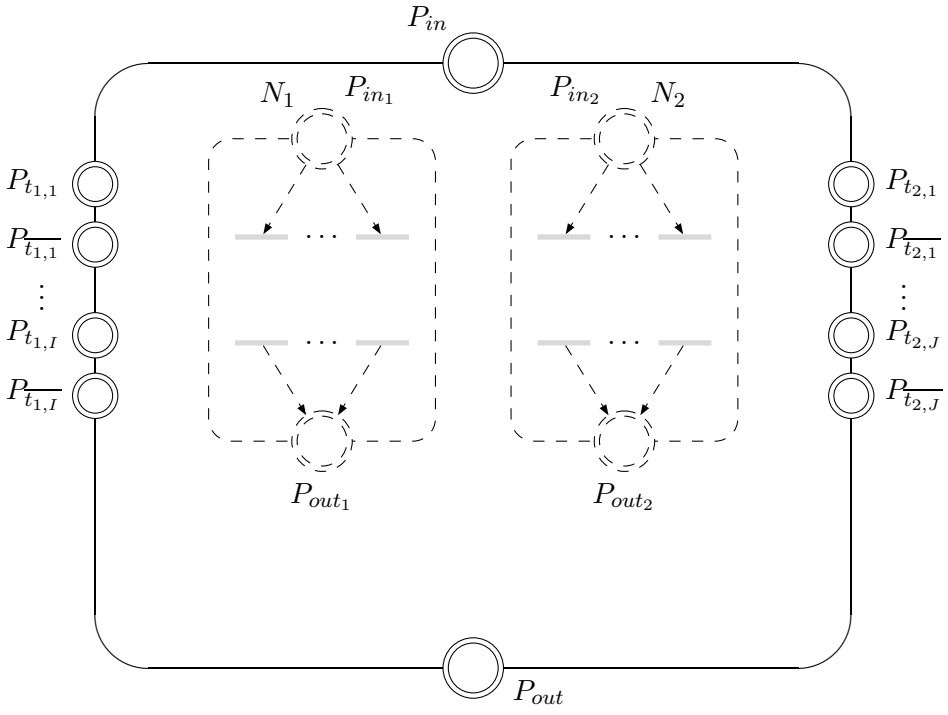


Figure 11.32: Tokens are visible outside

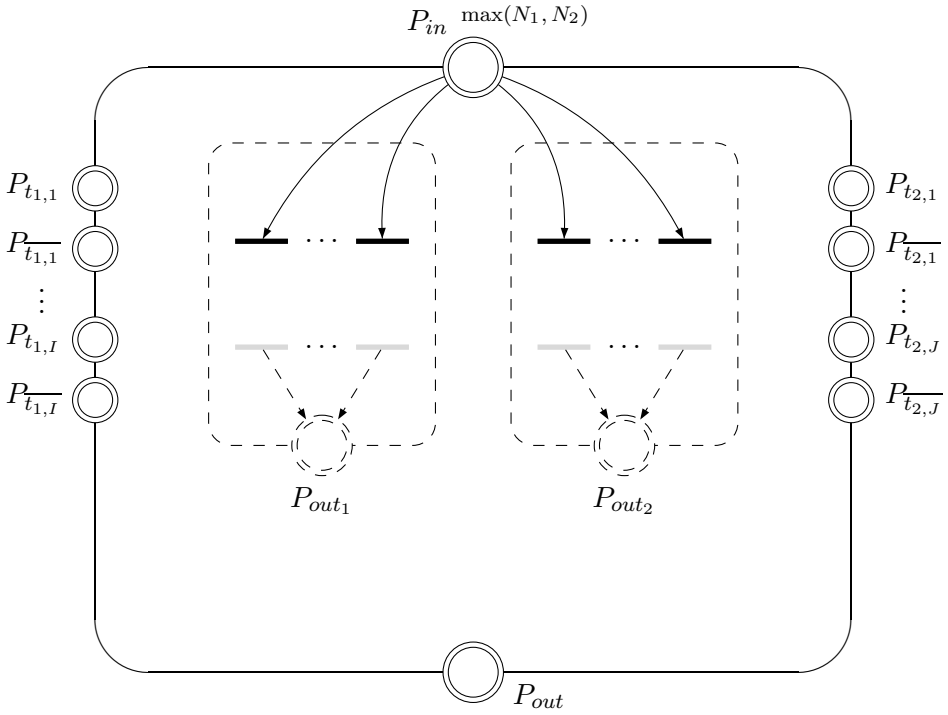


Figure 11.33: Entry place of choice agent connected to transitions

execution of the parallel agent is equal to the maximum between N_1 and N_2, i.e. $\max(N_1, N_2)$. Figure 11.33 shows the merging of the entry places, with the indication of the number of tokens.

The closure of the global agent must also reflects the choice construction. As only one of the two agents is effectively executed, only one of the two P_{out_1} and P_{out_2} places will provide its tokens to the exit place P_{Out}. This implies that P_{out_1} and P_{out_2} are standard (i.e. non-open) places of the composition and that two new transitions T_{out_1} and T_{out_2} are now introduced in the construction. Figure 11.34 shows the closure construction of the DB-Open Peti Net of the choice compositional agent.

In order to guarantee the exclusive selection of one of the two agents it is necessary to introduce a pair of internal auxiliary places P_{aux_1} and P_{aux_2} in the construction, each of these places being dedicated to one of the two subagents. These places serve to lock the choice between the two subagents. For a specific subagent, every transition connected to the entry place P_{in} is not only connected with its corresponding auxiliary place, but also with the auxiliary place associated with the second subagent. With such a construction, the execution of an agent depends, on the one hand, on the presence of the required number of tokens in its specific auxiliary place, but also, on the other hand, on the check whether the second agent has not been selected before. Then the firing of the transitions of one subagent has not only for effect

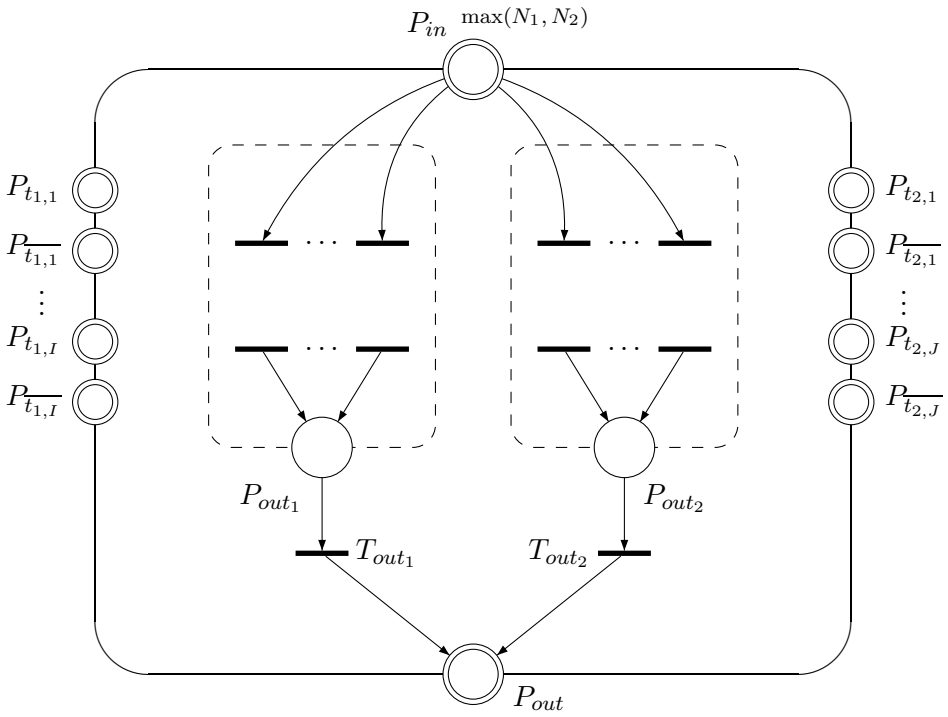


Figure 11.34: Connections to exit place of choice agent

to make its corresponding auxiliary place empty, but also to prevent the second subagent to be selected in the same execution of the choice. Regarding the initialization of an auxiliary place P_{aux_i}, it is done with the same number of tokens as in the corresponding entry place P_{in_i} of the concerned subagent, i.e. N_i. For the closure of the execution of the subagent, the firing of the corresponding T_{out} transition must restore the auxiliary place in its initial state. If the selected subagent is the one with the smallest required number on tokens, i.e. with N_2 as we supposed, this implies that $N_1 - N_2$ tokens are not consumed in the entry place P_{in}. In order to clean this place, a link will connect it with the transition T_{out_2}, weighted with the number $N_1 - N_2$. If both subagents require the same number of tokens for their execution, i.e. $N_1 = N_2$ this link is not necessary and is not taken into account. Figures 11.35 and 11.36 show respectively these constructions for the first and the second subagent.

Using again the notations of Definition 29, the resulting DB-open Petri Net associated with the choice composition $A_1 + A_2$ is the DB-open Petri Net $(S, T, {}^\bullet(.), (.)^\bullet, O, P_{in}, P_{out}, Pts, M, N)$ on $\mathcal{T}$ defined as follows:

- $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2$
- P_{in} is its (fresh) entry place
- P_{out} is its (fresh) exit place

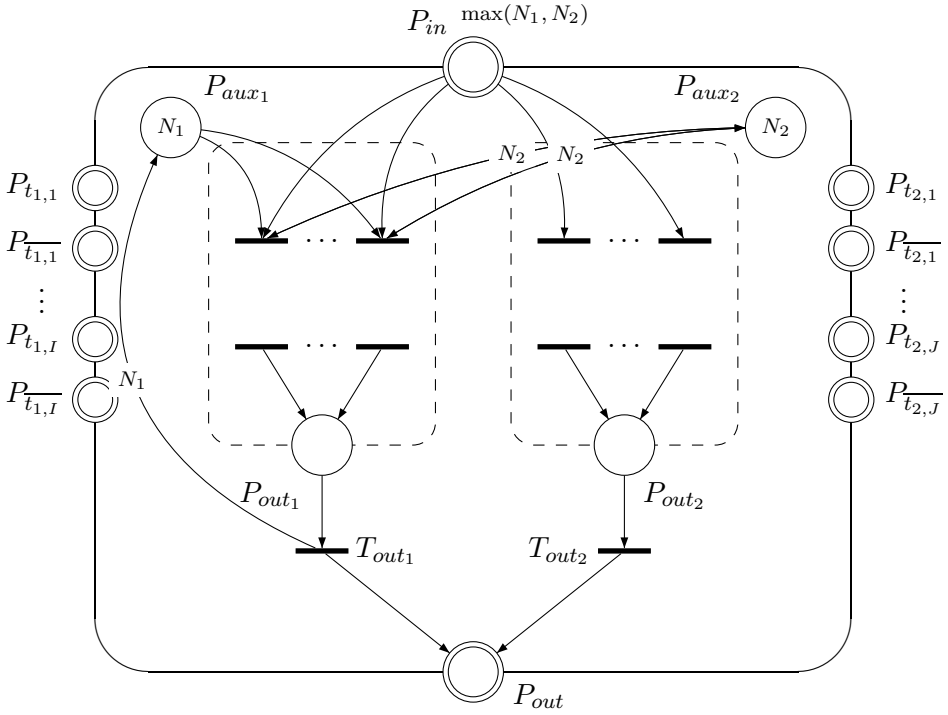


Figure 11.35: Connections to first auxiliary place

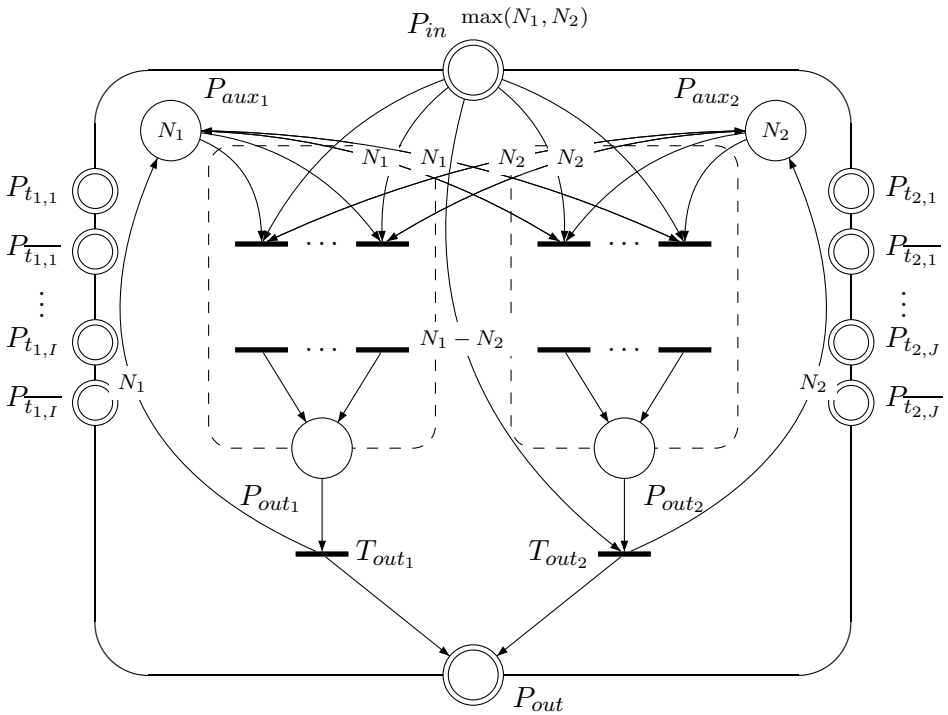


Figure 11.36: Connections to second auxiliary place

- Pts $= Pts_1 \cup Pts_2$
- $O = \{P_{in}, P_{out}\} \cup Pts$
- $S = (S_1 \backslash O_1) \cup (S_2 \backslash O_2) \cup O \cup \{P_{out_1}, P_{out_2}, P_{aux_1}, P_{aux_2}\}$
- $T = T_1 \cup T_2 \cup \{T_{out_1}, T_{out_2}\}$
- $^\bullet(T) = \begin{cases} {}^\bullet(T)_1 & where \quad \{P_{in_1}\} \quad is\ replaced\ by \quad \{P_{in}, P_{aux_1}, P_{aux_2}(N_2)\} \quad if \quad T \in T_1 \\ {}^\bullet(T)_2 & where \quad \{P_{in_2}\} \quad is\ replaced\ by \quad \{P_{in}, P_{aux_2}, P_{aux_1}(N_1)\} \quad if \quad T \in T_2 \end{cases}$
- $^\bullet(T_{out_1}) = \{P_{out_1}\}$
- $^\bullet(T_{out_2}) = \{P_{out_2}, P_{in}(N_1 - N_2)\}$
- $(T)^\bullet = \begin{cases} (T)_1^\bullet \cup \{P_{aux_2}(N_2)\} & if \quad T \in T_1 \quad and \quad P_{in} \in {}^\bullet(T) \\ (T)_1^\bullet & if \quad T \in T_1 \quad and \quad P_{in} \notin {}^\bullet(T) \\ (T)_2^\bullet \cup \{P_{aux_1}(N_1)\} & if \quad T \in T_2 \quad and \quad P_{in} \in {}^\bullet(T) \\ (T)_2^\bullet & if \quad T \in T_2 \quad and \quad P_{in} \notin {}^\bullet(T) \end{cases}$
- $(T_{out_1})^\bullet = \{P_{out}, P_{aux_1}(N_1)\}$
- $(T_{out_2})^\bullet = \{P_{out}, P_{aux_2}(N_2)\}$
- $M(P_t) = 0 \quad \forall t \in \mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2$
- $M(P_{\overline{t}}) = MAX \quad \forall \overline{t} \in \mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2$
- $M(P_{aux_1}) = N_1$
- $M(P_{aux_2}) = N_2$
- $M(P) = \begin{cases} M_1(P) & \text{for any } P \in S_1 \backslash O_1 \\ M_2(P) & \text{for any } P \in S_2 \backslash O_2 \end{cases}$
- $N = Max(N_1, N_2)$

11.4 Towards a workbench

One of the key features of Petri Nets is their property to depict suggestively computations. As a result, our translation of Dense Bach agents to Petri Nets allows to benefit from an expressive way of describing the executions of Dense Bach agents. We turn in this section to the development of a workbench based on these ideas.

As we can already take profit of a parser for Dense Bach from Chapter 8, the structure of our code consists in adding three components. The first one called *petriNetEquivalent.scala*

is in charge of the translation of the parsed expression to its equivallent Petri Net form. The second file called *densebach2petri_xml.scala* draws the different elements of the created Petri Net structure as a svg file. The third file called *runnigPetri.scala* is in charge of the dynamic execution of the Petri Net, showing its evolution, firing after firing of its transitions. It is to be noted that in our code the Petri Net has an extended form, to include the representation of the token space. The structure of the extended Petri Net itself is defined in a file of abstract data called *petriNetElement.scala*.

11.4.1 Main data structures

The petriNetElement.scala file defines two abstract classes. The first one, called *PnElmt*, defines the four elements constituing a Petri Net. The second one, called *PetriNet*, defines the Petri Net as a union of the previous four elements, extended with the representation of the token space.

The class *PnElmt* is refined in four classes, to represent the components of a Petri Net:

- a case class *place* to represent the places of the Petri Net. Many arguments are associated with a place. In particular, every place is represented with a circle, with two coordinates *cx* and *cy* for its center, and one *radius* for its radius. Three booleans indicate the nature of the place: an *entry* one, an *exit* one and a *regular* one. An entry place indicates the entry place of an agent. An exit place is the last place of an agent. Both these places are represented in blue in the net associated with the global agent. A regular place is an intermediate place inside the net, including the places of control of the execution, in the case of a parallel or a choice agent. The variable *nbrTokens* represents the number of tokens in the place. The variable *nbrReq* represents the minimum number of tokens that are to be present in the entry place of an agent in order for this one to be executed properly. Every place has an identifier represented by a list of strings *idplace*. This identifier is constructed recursively, to indicate to which level the place belongs. Finally, a string *caraGraphi* is used to represent the colour of the place.

- a case class *transition* to represent a transition between at least one incoming place and at least one outgoing place. A transition is represented by a rectangle, with four coordinates: the x and y coordinates of the upper left corner, the *width* of the rectangle and its *height*. The two strings *name* and *token* and the integer *density* indicates to which Dense Bach primitive the transition is related, and which dense token is concerned. For transitions that are not related to a Dense Bach primitive, those fields are set to the empty string

for the *name* and the *token* or to 0 for the *density*. The variable *idtrans* identifies the transition. As for the place, it has the form of a list of strings, and is built recursively. This permits to indicate to which level the transition is associated. Finally the string *caraGraphi* indicates the colour of the transition.

- a case class *pre_condition* to represent the pre_condition for the firing of a transition. Every pre_condition is characterised by an identifier of an incoming place *idplace*, an identifier of a transition *idtrans*, a *weight* and a graphical charateristic *caraGraphi*. The *weight* indicates the number of tokens that have to be consumed and retrieved in the associated incoming place by the firing of the transition.

- a case class *post_condition* to represent the post_condition after the firing of a transition. Every post_condition needs an identifier *idtrans* for the transition and an identifier *idplace* of an outgoing place. The integer *weight* indicates the number of tokens that are produced by the firing of the transition, and added to the corresponding outgoing place.

  The second abstract class *DbGraph* represents the extended Petri Net graph associated with a Dense Bach agent. It is refined in one class *PetriNet*, with six elements. The first four elements group the components of the previous *PnElmt* : a set of places *SetOfPlaces*, a set of transitions *SetOfTrans*, a set of pre_conditions *SetOfPre*, and a set of post_conditions *SetOfPost*. The last two elements represent the pre_conditions *SetOfPreSpaces* and post_conditions *SetOfPostSpaces* specific to the places of the token space.

The complete code of the abstract petriNetElement.scala file is depicted in Figure 11.37.

11.4.2 Converting Dense Bach agents to Petri Nets

Based on the result of the parsing of the agent, this file defines the class *PetriNetEquivalent*. It is mainly composed of one procedure *convert2petriNet*. The result of the parsing being a binary tree, this structure will be browsed by *convert2petriNet* in a recursive way, starting from the root of the tree, and progressively descending to the leaves that correspond to the primitives. Besides the parsed agent, this method also needs three other parameters : a *prefix* identification that represents the level of the concerned agent in the recursive structure, and two coordinates X and Y. These last two represent the coordinates of the upper left corner of the rectangle in which the agent will be drawn. The most basic agents - the primitives - find place in the smallest rectangle. When associated by a composition operator of choice or parallelism, their rectangles are included in the rectangle used to represent the composition. If associated by a sequential operator, their rectangles are sticked together.

```
abstract class PnElmt

case class place(cx : Int, cy : Int, radius : Int, entry : Boolean,
                                                  exit : Boolean, regular : Boolean,
     name : String, nbrTokens : Int, nbrReq : Int, idplace : List[String],
                                                 caraGraphi : String) extends PnElmt
case class transition(tx : Int, ty : Int, width : Int, height : Int,
                                                       name : String, token : String,
     density : Int, idtrans : List[String], caraGraphi : String) extends PnElmt
case class pre_condition(idplace : List[String], idtrans : List[String],
                                weight : Int, caraGraphi : String) extends PnElmt
case class post_condition(idtrans : List[String], idplace : List[String],
                                weight : Int, caraGraphi : String) extends PnElmt

abstract class DbGraph

case class PetriNet(SetOfPlaces : List[place], SetOfTrans : List[transition],
     SetOfPre : List[pre_condition], SetOfPost : List[post_condition],
     SetOfPreSpaces : List[pre_condition], SetOfPostSpaces : List[post_condition])
                                                                    extends DbGraph
```

Figure 11.37: The abstract petriNetElement.scala file

The result of the method *convert2petriNet* is a pair of information : the *Petri Net* associated with the treated agent, and an *integer* that represents the minimum number of tokens required for the execution of the Petri Net.

convert2petriNet makes use of many useful procedures to perform its action. They are not presented here, but the reader can find their code in annex (see Chapter G).

As said in the previous paragraph, every Petri Net construction is the result of a recursive process. It starts from the most complex form of parsed agent, and progressively adds its specific elements, i.e. places, transitions, and their pre and post_conditions, to the global Petri Net structure. The number and nature of added components depends on the way agents are combined. In the following, we first present the translation in Petri Net of the basic Dense Bach primitives. Then we will present the translation of a parsed agent that is the result firstly of a sequential composition of two agents, secondly of a parallel composition of two agents, and finally of a choice composition of two agents.

11.4.2.1 Basic primitives

The construction of the Petri Net representation of a primitive relies on the content of the expression *dbach_ast_primitive(primitive: String, token: String, density: Int, petricounter : Int, width : Int, height : Int)* provided by the parser. This expression is composed of six arguments.

The first one receives the name of the Dense Bach primitive : *tell*, *get*, *ask* or *nask*. The second variable receives the name of the token, and the third one its density. These three arguments are useful for the construction of the token space. The three last variables are more concerned by the Petri Net representation of the primitive. The first of these is an integer that represents the number of tokens that must be present in the entry place for a correct execution of the Petri Net. The two last variables are respectively the width and height of the rectangle within which the Petri Net representation of the primitive is to be drawn.

The construction starts with the definition of the X and Y coordinates of the centre of the entry place, *mean_X* and *mean_Y*. Their values are calculated on the basis of the X and Y coordinates, *posX* and *posY*, of the upper left corner of the rectangle within which the primitive is drawn. In particular the X coordinate is equal to the *posX* value, added with the half *width* of the rectangle :

```
val mean_X = posX + width/2
val mean_Y = posY + 10
```

Then the construction continues with the definition of the places of the token space, associated with the token of the primitive. As long as a token has not been registered on the token space, it is added on it. For every new token, two places are created. They respectively correspond to the space and anti-space represention of the token inside the token space. As the initial token space is empty, the place of the space receives zero token, and the place of the anti-space receives a number of *max_token* (defined in *densebach2petri_data.scala*). The code is the following, where *lToken* is the list of tokens already registered in the token space :

```
if(!(lToken.contains(token))) { // if token not yet register,
                                // create its place and antiplace
tokSpace = place(4*extra_w_center,3*radius/2,radius,false,false,
                  true,token,0,1,"ns"::token::Nil,"black")
aTokSpace = place(4*extra_w_center,3*radius/2,radius,false,false,
                  true,token,max_token,1,"as"::token::Nil,"red")
lToken :::= List(token)
}
```

After the token space, the construction continues with the creation of the entry *PrimIN* and exit *PrimOUT* places, and the transition *PrimTr* between them. The places are positioned on the vertical symetrical axis of the rectangle, i.e. with an X coordinate equal to *mean_X*. The rectangle of the transition has an upper left X coordinate at a distance *radius* of the symetrical axis. This value is defined in *densebach2petri_data.scala* and is equal to the half width of the

rectangle representing the transition. Finally, the pre_ and post_condition for the firing of the transition *PrimTr* are defined, with a weight of 1 token respectivelly. Those different elements are added to the different sets constituing a Petri Net: the Set of places, the set of transitions, the set of pre_conditions and the set of post_conditions.

```
val PrimIN  = place(mean_X,mean_Y,radius,true,false,false,"",
                  petricounter,1,"In"::prefid,"blue") // place IN
val PrimOUT = place(mean_X,mean_Y + height,radius,false,true,false,"",
                         0,1,"Out"::prefid,"blue")  // place OUT
val PrimTr  = transition(mean_X - radius,mean_Y + height/2,2*radius,radius/3,
               prim,token,density,"Tr"::prefid,"black") // transition Tr
val PrimPc  = pre_condition("In"::prefid,"Tr"::prefid,petricounter,"black")
                    // pre of Tr coming from IN
val PrimPo  = post_condition("Tr"::prefid,"Out"::prefid,petricounter,"black")
                    // post from Tr going to OUT
lPlace = PrimIN::PrimOUT::tokSpace::aTokSpace::pn.SetOfPlaces
                   // list of places associted with primitive
```

The pre and post_conditions relative to the places of the token space are then defined, based on the type of the primitive. For a *tell* primitive, the pre_condition must start from the anti-space. This is noted in the identifier of the pre_condition, formed by the string *as* - for anti-space - followed by the name of the *token*. The corresponding post_condition must go from the transition to the place in the token space. This is noted by the identifier of the post_condition, which is now noted as *ns* – for normal space – followed by the name of the *token*.

```
prim match { // pre and post following type of transition
 case "tell" => {%
  //for tell : from antispace (as) ...
PrimPcS = pre_condition("as"::token::Nil,"Tr"::prefid,density,"black")
// ... to normal space (ns)
PrimPoS = post_condition("Tr"::prefid,"ns"::token::Nil,density,"black")
 }
 case "get" => {
  // for get : from ns ...
PrimPcS = pre_condition("ns"::token::Nil,"Tr"::prefid,density,"black")
// ... to as
PrimPoS = post_condition("Tr"::prefid,"as"::token::Nil,density,"black")
 }
 case "ask" => {
  // for ask : from ns ...
PrimPcS = pre_condition("ns"::token::Nil,"Tr"::prefid,density,"black")
// ... to ns
```

```
PrimPoS = post_condition("Tr"::prefid,"ns"::token::Nil,density,"black")
 }
 case "nask" => {
  // for nask : from as ...
PrimPcS = pre_condition("as"::token::Nil,"Tr"::prefid,density,"black")
// ... to as
PrimPoS = post_condition("Tr"::prefid,"as"::token::Nil,density,"black")
 }
}
```

Finally, all the components produced are grouped with the existing components of the Petri Net. This one is returned as the result of the call to the method *convert2petriNet*.

```
(PetriNet( lPlace,
   PrimTr::pn.SetOfTrans,
   PrimPc::pn.SetOfPre,
   PrimPo::pn.SetOfPost,
   PrimPcS::pn.SetOfPreSpaces,
   PrimPoS::pn.SetOfPostSpaces),
petricounter) // synthesis of translation in PetriNet
```

The complete code for the construction of the Petri Net elements corresponding to a basic primitive is listed in Figures 11.38 and 11.39.

11.4.2.2 Sequential composition

The result of the parsing can produce three kinds of complex agents. The first of these one corresponds to the sequential composition of two agents. In that case, the result of the parsing is an expression of the form *dbach_ast_agent(";",ag_i ,ag_ii ,petricounter,width,height)* that contains six parameters. The first one is a semi-colon that represents the sequential composition. The two next parameters represent the parsed subagents *ag_i* and *ag_ii* that are sequentially combined. The fourth parameter is the number of tokens needed by the Petri Net for a correct execution of the sequential composition, as it is calculated in the parsing of a sequential agent. This number is equal to the petricounter of the first agent *ag_i* of the composition. The two last parameters are the width and the height of the rectangle that contains the drawing of the sequential composition. The Petri Net construction makes a recursive call to the *convert2petriNet* method in order to build the representation of the two sub-agents *ag_i* and *ag_ii*. The recursive call specifies the level of the call, adding 1 to the prefix id of the first agent, and 2 to the one of the second agent.

```
case dbach_ast_primitive(prim,token,density,petricounter,width,height) => {
 // primitive to Petri Net
 val mean_X = posX + width/2
 val mean_Y = posY + 10

 // place associated with token in token space
 var tokSpace = place(0,0,0,false,false,false,"",0,1,Nil,"")
 // antiplace associated with token in token space
 var aTokSpace = place(0,0,0,false,false,false,"",0,1,Nil,"")
 // pre condition of transition Tr (defined forward) coming from place tokSpace
 var PrimPcS = pre_condition(Nil,Nil,0,"")
 // post condition of transition Tr (defined forward) going to place aTokSpace
 var PrimPoS = post_condition(Nil,Nil,0,"")

 // create the place and anti-place in token space for a new token
 // (not yet registered in list of lToken)

 if(!(lToken.contains(token))) {
  // if token not yet register, create its place and antiplace
  tokSpace = place(4*extra_w_center,3*radius/2,radius,false,false,true,token,
                                                          0,1,"ns"::token::Nil,"black")
  aTokSpace = place(4*extra_w_center,3*radius/2,radius,false,false,true,token,
                                                  max_token,1,"as"::token::Nil,"red")
  lToken :::= List(token)
 }

 // place IN
 val PrimIN  = place(mean_X,mean_Y,radius,true,false,false,"",petricounter,
                                                          1,"In"::prefid,"blue")
 // place OUT
 val PrimOUT = place(mean_X,mean_Y + height,radius,false,true,false,"",
                                                          0,1,"Out"::prefid,"blue")
 // transition Tr
 val PrimTr  = transition(mean_X - radius,mean_Y + height/2,2*radius,radius/3,prim,
                                                  token,density,"Tr"::prefid,"black")
 // pre of Tr coming from IN
 val PrimPc  = pre_condition("In"::prefid,"Tr"::prefid,petricounter,"black")
```

Figure 11.38: The code for the construction of the Petri Net elements of a basic primitive

```
// post from Tr going to OUT
 val PrimPo  = post_condition("Tr"::prefid,"Out"::prefid,petricounter,"black")
 var lPlace : List[place] = Nil
 var lPlace1: List[place] = Nil

 lPlace = PrimIN::PrimOUT::tokSpace::aTokSpace::pn.SetOfPlaces
 // list of places associted with primitive
 lPlace = removeEmptyPlace(lPlace)

 prim match { // pre and post following type of transition
  case "tell" => {
    //for tell : from antispace (as) ...
    PrimPcS = pre_condition("as"::token::Nil,"Tr"::prefid,density,"black")
    // ... to normal space (ns)
    PrimPoS = post_condition("Tr"::prefid,"ns"::token::Nil,density,"black")
   }
  case "get" => {
   // for get : from ns ...
   PrimPcS = pre_condition("ns"::token::Nil,"Tr"::prefid,density,"black")
   // ... to as
   PrimPoS = post_condition("Tr"::prefid,"as"::token::Nil,density,"black")
  }
  case "ask" => {
   // for ask : from ns ...
   PrimPcS = pre_condition("ns"::token::Nil,"Tr"::prefid,density,"black")
   // ... to ns
   PrimPoS = post_condition("Tr"::prefid,"ns"::token::Nil,density,"black")
  }
  case "nask" => {
   // for nask : from as ...
   PrimPcS = pre_condition("as"::token::Nil,"Tr"::prefid,density,"black")
   // ... to as
   PrimPoS = post_condition("Tr"::prefid,"as"::token::Nil,density,"black")
  }
 }

 (PetriNet( lPlace,
        PrimTr::pn.SetOfTrans,
        PrimPc::pn.SetOfPre,
        PrimPo::pn.SetOfPost,
        PrimPcS::pn.SetOfPreSpaces,
        PrimPoS::pn.SetOfPostSpaces),
        petricounter) // synthesis of translation in PetriNet
}
```

Figure 11.39: The code for the construction of the Petri Net elements of a basic primitive (cont)

```
case dbach_ast_agent(";",ag_i,ag_ii,petricounter,width,height) => {
 if (ag_i.width == ag_ii.width) {
  val (pn1,pc1) = convert2petriNet(ag_i,"1"::prefid,posX +
                                                    ag_i.width/2,posY)
  val (pn2,pc2) = convert2petriNet(ag_ii,"2"::prefid,posX +
                                    ag_i.width/2,posY + ag_i.height)
  ...
  }
```

The sequential composition must take into account the width of the two rectangles that are to be connected. Depending on their respective width, the construction will be done in order to align their respective vertical axis.

```
if (ag_i.width == ag_ii.width) {
 ...
} else if (ag_i.width > ag_ii.width) {
 ...
} else {
 ...
}
```

When both agents are constructed, the fusion between them is done by replacing the entry place of the second one by the exit place of the first one. At this level, the number of tokens of this exit place must be equal to the tokens associated with the Petri Net of the second agent. Finally, the different components resulting from those recursive calls are grouped in the Petri Net structure returned by the *convert2petriNet* method.

```
if (ag_i.width == ag_ii.width) {
 val (pn1,pc1) = convert2petriNet(ag_i,"1"::prefid,posX +
                                                 ag_i.width/2,posY)
 val (pn2,pc2) = convert2petriNet(ag_ii,"2"::prefid,posX +
                                  ag_i.width/2,posY + ag_i.height)
 val pn1seq = PetriNet(transfOutToRegPlace(pn1.SetOfPlaces,pc2),
                pn1.SetOfTrans,pn1.SetOfPre,pn1.SetOfPost,
                           pn1.SetOfPreSpaces,pn1.SetOfPostSpaces)
     // replace In from ag_ii by Out from ag_i
 val pn2seq = removeInAndReplace(findOutPlace(pn1.SetOfPlaces),1,pn2)
 (unionPetriNet(pn1seq,pn2seq),pc1)
} else if ...
```

The complete code for the construction of the Petri Net elements of a sequential composition of two agents is depicted in Figure 11.40.

```
case dbach_ast_agent(";",ag_i,ag_ii,petricounter,width,height) => {

 // sequence of agents to Petri Net
 if (ag_i.width == ag_ii.width) {
    // if ag_i and ag_ii have SAME width
  val (pn1,pc1) = convert2petriNet(ag_i,"1"::prefid,posX + ag_i.width/2,posY)
  val (pn2,pc2) = convert2petriNet(ag_ii,"2"::prefid,posX + ag_i.width/2,
                                                          posY + ag_i.height)
  val pn1seq = PetriNet(transfOutToRegPlace(pn1.SetOfPlaces,pc2),pn1.SetOfTrans,
              pn1.SetOfPre,pn1.SetOfPost,pn1.SetOfPreSpaces,pn1.SetOfPostSpaces)
    // replace In from ag_ii by Out from ag_i
  val pn2seq = removeInAndReplace(findOutPlace(pn1.SetOfPlaces),1,pn2)
  (unionPetriNet(pn1seq,pn2seq),pc1)

 } else if (ag_i.width > ag_ii.width) {
    // if ag_i is LARGER than ag_ii
  val (pn1,pc1) = convert2petriNet(ag_i,"1"::prefid,posX + ag_i.width/2,posY)
  val (pn2,pc2) = convert2petriNet(ag_ii,"2"::prefid,posX + ag_i.width/2 - ag_ii.width/2,
                                                          posY + ag_i.height)
  val pn1seq = PetriNet(transfOutToRegPlace(pn1.SetOfPlaces,pc2),pn1.SetOfTrans,
                       pn1.SetOfPre,pn1.SetOfPost,pn1.SetOfPreSpaces,pn1.SetOfPostSpaces)
  val pn2seq = removeInAndReplace(findOutPlace(pn1.SetOfPlaces),1,pn2)
  (unionPetriNet(pn1seq,pn2seq),pc1)

 } else {
   // if ag_i is SMALLER than ag_ii
  val (pn1,pc1) = convert2petriNet(ag_i,"1"::prefid,
                                          posX + ag_ii.width/2 - ag_i.width/2,posY)
  val (pn2,pc2) = convert2petriNet(ag_ii,"2"::prefid,posX + ag_i.width,
                                                          posY + ag_i.height)
  val pn1seq = PetriNet(transfOutToRegPlace(pn1.SetOfPlaces,pc2),pn1.SetOfTrans,
                      pn1.SetOfPre,pn1.SetOfPost,pn1.SetOfPreSpaces,pn1.SetOfPostSpaces)
  val pn2seq = removeInAndReplace(findOutPlace(pn1.SetOfPlaces),1,pn2)
  (unionPetriNet(pn1seq,pn2seq),pc1)
 }
}
```

Figure 11.40: The code for the construction of the Petri Net elements of a sequential composition of two agents

11.4.2.3 Parallel composition

In case the complex agent corresponds to a parallel composition of two sub-agents, the construction is based on the resulting expression *dbach_ast_agent(" || ",ag_i,ag_ii,petricounter,width,height)* of the parsing. As for the sequential composition, it contains six parameters. The first one is a string representing the parallel composition. The two next parameters represent the parsed agents *ag_i* and *ag_ii* that are combined in parallel. The fourth parameter is the number of tokens of the Petri Net structure for the parallel composition, as it is calculated in the parsing

of the parallel agent. This number is equal to the sum of the petricounters of *ag_i* and *ag_ii*. The two last parameters are the width and the height of the rectangle that contains the drawing of the parallel composition.

As for a primitive, the construction starts with the definition of the X and Y coordinates of the centre of the entry place, *mean_X* and *mean_Y*. Their values are calculated on the basis of the X and Y coordinates, *posX* and *posY*, of the upper left corner of the rectangle within which the primitive will be drawn. In particular the X coordinate is equal to the *posX* value, added with the half *width* of the rectangle :

```
val mean_X = posX + width/2
val mean_Y = posY + 10
```

The first step is then to call the recursive procedure *convert2petriNet* for the construction of the parsed agent *ag_i* and *ag_ii*. Every agent receives a prefix id, which is the prefix id associated with the parallel construction, increased by 1 for the first agent, and 2 for the second agent. These prefix id are used for the construction of the places and transitions of a specific agent.

```
       // pn1 associated with ag_i
val (pn1,pc1)  = convert2petriNet(ag_i,"1"::prefid,mean_X - extra_w_center/2 -
                 ag_i.width,mean_Y + 2*extra_h_top/3)
       // pn2 associated with ag_ii
val (pn2,pc2)  = convert2petriNet(ag_ii,"2"::prefid,mean_X + extra_w_center/2,
                                                    mean_Y + 2*extra_h_top/3)
```

For their construction, the X and Y coordinates of the upper left corner of the rectangle including them must be provided. They are calculated based on the representation of the arrangement of the rectangle of both agents inside the global rectangle of the parallel composition, as seen in Figure 11.41. For instance, based on this schema the X coordinate of the upper left corner of agent *ag_i* on the left part of the picture is obtained by substracting from the mean position *mean_X* the value of half the constant *extra_w_center* and the width of the rectangle of *ag_i.width*. The values of the different constants *extra_w_center*, *extra_w_right*, *extra_w_left*, *extra_h_top* and *extra_h_bottom* are defined in the file *densebach2petri_data.scala*. *radius* is a constant used in many calculations, as an adjusting value. It is also half the width of the rectangle representing a transition.

The entry and exit places of the parallel agent are also represented in Figure 11.41. These places are positioned onto the symetric vertical axis, and receive an identifier with a prefix

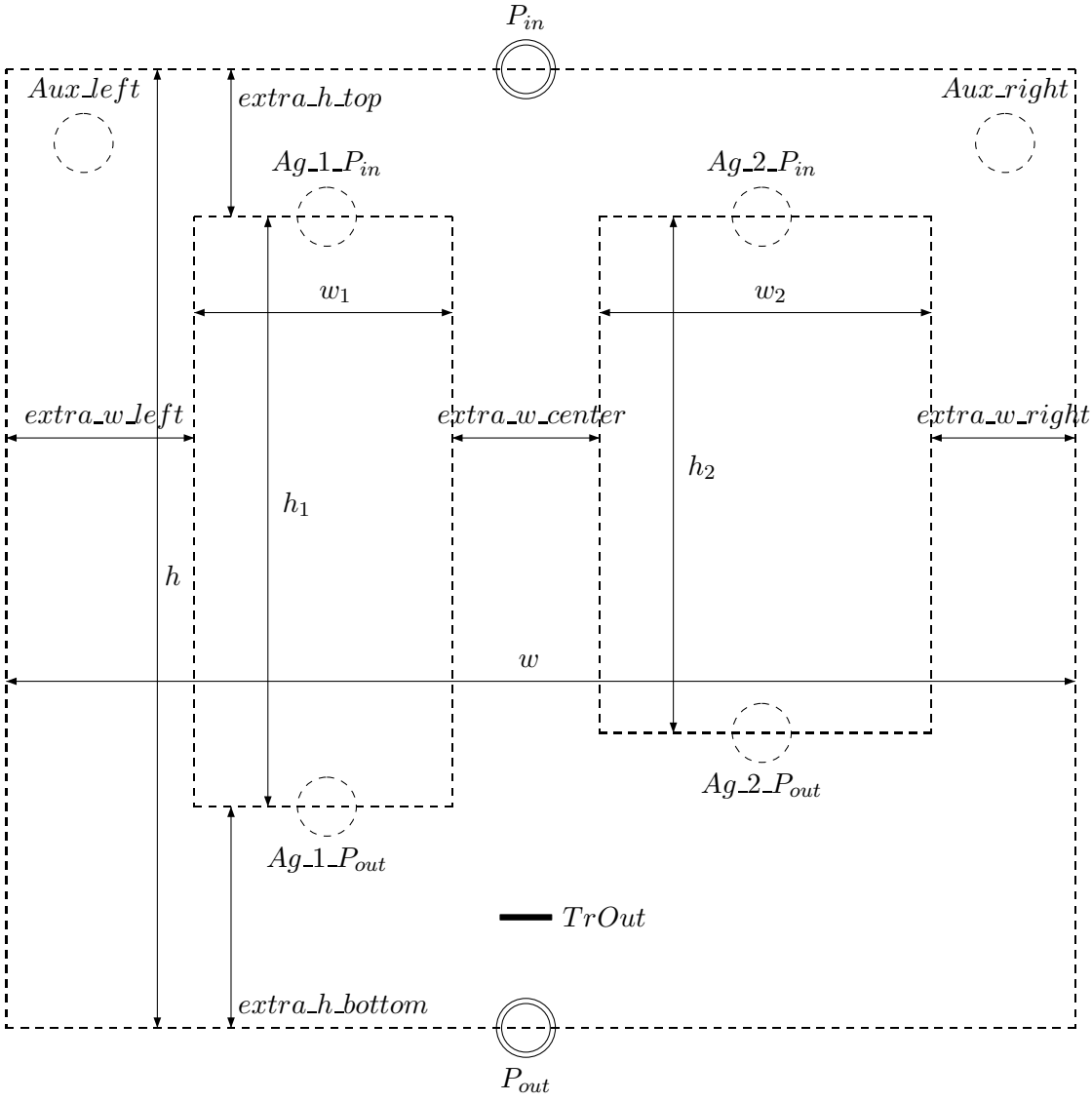


Figure 11.41: The schema of building of a complex agent based on two agents ag_1 and ag_2 for a parallel composition

id coresponding to the level of the recursive call. The number of tokens in the entry place is equal to the sum of the numbers of tokens of the two sub-Petri Nets corresponding to the two sub-agents. The code defining these places is as follows:

```
val In = place(mean_X,mean_Y,radius,true,false,false,"",
               valueSum(pc1,pc2),valueSum(pc1,pc2),"In"::prefid,"blue")
val Out  = place(mean_X,mean_Y + height,radius,false,true,
               false,"",0,1,"Out"::prefid,"blue")
```

The construction has also to take into account the addition of some extra elements in the resulting Petri Net. These elements are essentially places of control, called *auxiliary places*. Their function is to regulate the execution of the sub-agents. In a parallel composition, every sub-agent must be selected one time. As the entry place of the parallel agent contains exactly the number of tokens that is necessary for selecting both sub-agent, there is a risk that a same sub-agent could be selected more than one time. In order to avoid this, as evidenced in section 11.3.2.2, two control places are added to the representation of the Petri Net. They are called the *left auxiliary place* and the *right auxiliary place*, left and right with regard to the vertical symetrical axis. They are respectively responsible for locking the selection of the corresponding agent situated on the same side once it has been fired. The code defining the auxiliary places is depicted hereafter. The number of tokens these auxiliary places will receive is the one contained in the entry place of the sub-agent they control. This can depend on the type of the sub-agents, essentially if it results from a sequential composition or not. This difference can be expressed by the difference between the length of the list id of the respective entry places of the parallel agent and the length of the list id of the entry place of the sub-agent. Following this, the entry place to be invoked for obtaining its number of tokens has a different id. This discussion is presented into the following code, with the definition of the auxiliary places *Auxg* and *Auxd*.

```
if (findInPlace(pn1.SetOfPlaces).length - In.idplace.length == 1) {
 Auxg = place(mean_X - width/2 + radius,mean_Y + extra_h_top - 2*radius,
         radius,false,false,true,"",findWeightInPlace("In"::"1"::prefid,
                                pn1.SetOfPlaces),1,"Auxl"::prefid,"black")
} else {
 Auxg = place(mean_X - width/2 + radius,mean_Y + extra_h_top - 2*radius,
         radius,false,false,true,"",findWeightInPlace("In"::"1"::"1"::prefid,
                                 pn1.SetOfPlaces),1,"Auxl"::prefid,"black")
}
if (findInPlace(pn2.SetOfPlaces).length - In.idplace.length == 1) {
 Auxd = place(mean_X + width/2 - radius,mean_Y + extra_h_top - 2*radius,
```

```
        radius,false,false,true,"",findWeightInPlace("In"::"2"::prefid,
                                pn2.SetOfPlaces),1,"Auxr"::prefid,"black")
} else {
 Auxd = place(mean_X + width/2 - radius,mean_Y + extra_h_top - 2*radius,
        radius,false,false,true,"",findWeightInPlace("In"::"1"::"2"::prefid,
                                pn2.SetOfPlaces),1,"Auxr"::prefid,"black")
}
```

The auxiliary places have to control all the possible agents that composed a sub-agent. For all of them it is then necessary to define the pre_conditions between the auxiliary place and the concerned transitions. This is done by the following code :

```
        // build all the pre of Auxg of parallel agent
PrePn1Aux = giveAll("Auxl"::prefid, PrePn1In, 1, "black")
        // idem for Auxd
PrePn2Aux = giveAll("Auxr"::prefid, PrePn2In, 1, "black")
```

The pre_conditions of the entry place for both sub-agents *PrePn1In* and *PrePn2In* are the result of the procedure *collectIdTransPre* applied to the respective set of pre_conditions, for the entry place, like for instance into the following code :

```
PrePn1Temp1 = collectIdTransPre(pn1.SetOfPre, "In"::"1"::"1"::prefid)
```

The pre_conditions of the left auxiliary place *Auxl* are constructed and collected in *PrePn1Aux*. Every transition will receive a weigth of 1, and the color of the arrow will be black. The same code applies for the right auxiliary place *Auxr*.

The insertion of both sub-agents in the parallel structure implies a replacement of their respective entry places by the entry place of the parallel composition. This implies also a modification of the pre_conditions related to these entries places, where the id of the entry place must now be one of the entry place of the parallel composition. The following code is responsible for these modifications :

```
var pn11 = removeInAndReplace("In"::prefid,1,pn1)
var pn22 = removeInAndReplace("In"::prefid,1,pn2)
```

After the firing of the sub-agents, a token is present in their own exit places. These sub-results must be grouped within the exit place of the parallel construction. This is done by one transition that will be fired if all the exit places of the sub-agents are populated with tokens. The result is then the production of one token in the exit place of the parallel construction.

This schema corresponds to our model of basic brick for the construction of a Petri Net: on the one hand, one entry place is followed by a list of transitions, and, on the other hand, some exit transitions opening to one final exit place. The code for building this transition is as follows :

```
val To = transition(mean_X - radius,mean_Y + height - 2*extra_h_bottom/3,
                          2*radius,radius/3,"To","",0,"To"::prefid,"black")
```

This transition *To* is a rectangle where the X and Y coordinates of its upper left corner are calculated based on the constants *radius* and *extra_h_bottom* of Figure 11.41. This transition has pre_conditions, established with the exit places of both sub-agents. These pre_conditions have a weight of 1 and their color is black. Their definitions are given by the following code :

```
val OutgTo = pre_condition(findOutPlace(pn11.SetOfPlaces),"To"::prefid,1,"black")
val OutdTo = pre_condition(findOutPlace(pn22.SetOfPlaces),"To"::prefid,1,"black")
```

It is to be noted that the firing of this transition *To* implies to restore the auxiliary places in their original state. This implies to define post_conditions between the transition *To* and both auxiliary places *Auxl* and *Auxr*. It is also necessary to define the post_condition between the transition *To* and the final exit place *out*. The code is the following :

```
val ToAuxg    = post_condition("To"::prefid,"Auxl"::prefid,
                              findWeightInPlace("Auxl"::prefid, Auxg::Nil),"black")
val ToAuxd    = post_condition("To"::prefid,"Auxr"::prefid,
                              findWeightInPlace("Auxr"::prefid, Auxd::Nil),"black")
val ToOut     = post_condition("To"::prefid,"Out"::prefid,1,"black")
```

Restoring the auxiliary place allows our code to be used to process agents with an iterative structure. Nevertheless that possibility has not been developped further in our thesis.

Finally, all the components produced are grouped in the Petri Net structure, as follows:

```
(PetriNet(In::Auxg::Auxd::Out::pn11.SetOfPlaces:::pn22.SetOfPlaces,
 To::pn11.SetOfTrans:::pn22.SetOfTrans,
 OutgTo::OutdTo::Pre1:::Pre2,
 ToAuxg::ToAuxd::ToOut::pn11.SetOfPost:::pn22.SetOfPost,
    pn1.SetOfPreSpaces:::pn2.SetOfPreSpaces,
    pn1.SetOfPostSpaces:::pn2.SetOfPostSpaces),
    valueSum(pc1,pc2))
```

Figures 11.42 to 11.44 depict the complete code for the construction of the Petri Net element in the case of a parallel composition of two agents.

```
case dbach_ast_agent("||",ag_i,ag_ii,pc,width,height) => {
    // parallel of agents ag_i and ag_ii to Petri Net

    /*
    build places In, Auxg (auxiliary left place), Auxd (auxiliary right place) and Out;
    build transition To
    remove places In of ag_i and ag_ii, and remplace them by In (of parallel agent);
    modify related pre (of previous action)
    build pre of Auxg to transition of ag_i and pre of Auxd to transition of ag_ii;
    build post of TO to Auxg, Auxd and Out
    */

    val mean_X = posX + width/2
    val mean_Y = posY + 10

    val (pn1,pc1)  = convert2petriNet(ag_i,"1"::prefid,mean_X - extra_w_center/2
                     - ag_i.width,mean_Y + 2*extra_h_top/3) // pn1 associated with ag_i
    val (pn2,pc2)  = convert2petriNet(ag_ii,"2"::prefid,mean_X + extra_w_center/2,
                                mean_Y + 2*extra_h_top/3) // pn2 associated with ag_ii

    // create In with number of tokens equal to summation of pc1 and pc2
    val In = place(mean_X,mean_Y,radius,true,false,false,"",valueSum(pc1,pc2),
                                                valueSum(pc1,pc2),"In"::prefid,"blue")
    val Out  = place(mean_X,mean_Y + height,radius,false,true,false,"",0,1,
                                                                "Out"::prefid,"blue")
    var Auxg = place(0,0,0,false,false,false,"",0,0,Nil,"")
    var Auxd = place(0,0,0,false,false,false,"",0,0,Nil,"")

    // To take into account the sequentiality of two agents
    if (findInPlace(pn1.SetOfPlaces).length - In.idplace.length == 1) {
     Auxg = place(mean_X - width/2 + radius,mean_Y + extra_h_top - 2*radius,radius,
        false,false,true,"",findWeightInPlace("In"::"1"::prefid, pn1.SetOfPlaces),1,
                                                                "Auxl"::prefid,"black")
    } else {
     Auxg = place(mean_X - width/2 + radius,mean_Y + extra_h_top - 2*radius,radius,
        false,false,true,"",findWeightInPlace("In"::"1"::"1"::prefid, pn1.SetOfPlaces),
                                                                1,"Auxl"::prefid,"black")
    }
    if (findInPlace(pn2.SetOfPlaces).length - In.idplace.length == 1) {
     Auxd = place(mean_X + width/2 - radius,mean_Y + extra_h_top - 2*radius,radius,
        false,false,true,"",findWeightInPlace("In"::"2"::prefid, pn2.SetOfPlaces),1,
                                                                "Auxr"::prefid,"black")
    } else {
     Auxd = place(mean_X + width/2 - radius,mean_Y + extra_h_top - 2*radius,radius,
        false,false,true,"",findWeightInPlace("In"::"1"::"2"::prefid, pn2.SetOfPlaces),
                                                                1,"Auxr"::prefid,"black")
    }
```

Figure 11.42: The code for the construction of the Petri Net elements of a parallel composition of two agents

```
var PrePn1Temp1      : List[List[String]]   = Nil //
    var PrePn1Temp11     : List[List[String]]   = Nil //
    var PrePn1In         : List[List[String]]   = Nil // list of idtrans for pn1
    var PrePn2Temp2      : List[List[String]]   = Nil
    var PrePn2Temp22     : List[List[String]]   = Nil
    var PrePn2In         : List[List[String]]   = Nil // list of idtrans for pn2
    var PrePn1Aux        : List[pre_condition]  = Nil // list of pre of aux place for pn1
    var PrePn2Aux        : List[pre_condition]  = Nil // list of pre of aux place for pn2
    var PrePn1Aux1       : List[pre_condition]  = Nil // list of pre of place aux1 for pn1
    var PrePn2Aux2       : List[pre_condition]  = Nil // list of pre of place aux2 for pn2
    var Pre1             : List[pre_condition]  = Nil
    var Pre2             : List[pre_condition]  = Nil

    if(findInPlace(pn1.SetOfPlaces).length > 3) { // if ag_i is a composition of agents
     PrePn1Temp1 = collectIdTransPre(pn1.SetOfPre, "In"::"1"::"1"::prefid)
    } else {
       // collect idtrans in Pre associated with place IN::1 of pn1 (= of ag_i)
     PrePn1Temp1 = collectIdTransPre(pn1.SetOfPre, "In"::"1"::prefid)
    }
    if(findInPlace(pn2.SetOfPlaces).length > 3) { // if ag_ii is a composition of agents
     PrePn2Temp2 = collectIdTransPre(pn2.SetOfPre, "In"::"1"::"2"::prefid)
    } else {
     PrePn2Temp2 = collectIdTransPre(pn2.SetOfPre, "In"::"2"::prefid)
    }

    if(findInPlace(pn1.SetOfPlaces).length > 2) { // if ag_ii is a composition of agents
     PrePn1Temp11 = collectIdTransPre(pn1.SetOfPre, "In"::"1"::"1"::prefid)
    } else {
      // collect idtrans in Pre associated with place IN::1 of pn1 (= of ag_i)
     PrePn1Temp11 = collectIdTransPre(pn1.SetOfPre, "In"::"1"::prefid)
    }
    if(findInPlace(pn2.SetOfPlaces).length > 2) { // if ag_ii is a composition of agents
     PrePn2Temp22 = collectIdTransPre(pn2.SetOfPre, "In"::"1"::"2"::prefid)
    } else {
     PrePn2Temp22 = collectIdTransPre(pn2.SetOfPre, "In"::"2"::prefid)
    }

    if(PrePn1Temp1 == Nil) { // combine previous results
     PrePn1In = PrePn1Temp11
    } else {
     PrePn1In = PrePn1Temp1:::PrePn1Temp11
    }
    if(PrePn2Temp2 == Nil) {
     PrePn2In = PrePn2Temp22
    } else {
     PrePn2In = PrePn2Temp2:::PrePn2Temp22
    }
```

Figure 11.43: The code for the construction of the Petri Net elements of a parallel composition of two agents (cont)

```
PrePn1Aux = giveAll("Auxl"::prefid, PrePn1In, 1, "black") // build all the pre of Auxg
    PrePn2Aux = giveAll("Auxr"::prefid, PrePn2In, 1, "black") // idem for Auxd

    var pn11 = removeInAndReplace("In"::prefid,1,pn1) // remove IN::1 of ag_i, replaced
                                                      // by IN::prefid
    var pn22 = removeInAndReplace("In"::prefid,1,pn2) // idem for ag_ii

    PrePn1Aux1 = PrePn1Aux:::pn11.SetOfPre // regroup pre of aux for pn1 with pn11
    PrePn2Aux2 = PrePn2Aux:::pn22.SetOfPre // idem for pn2 with pn22

    Pre1 = PrePn1Aux1.distinct
    Pre2 = PrePn2Aux2.distinct

    val To = transition(mean_X - radius,mean_Y + height - 2*extra_h_bottom/3,2*radius,
                                                    radius/3,"To","",0,"To"::prefid,"black")

      // build the pre of places Outg and Outd with transition To
    val OutgTo    = pre_condition(findOutPlace(pn11.SetOfPlaces),"To"::prefid,1,"black")
    val OutdTo    = pre_condition(findOutPlace(pn22.SetOfPlaces),"To"::prefid,1,"black")

     // build the post of transition To
    val ToAuxg    = post_condition("To"::prefid,"Auxl"::prefid,
                                  findWeightInPlace("Auxl"::prefid, Auxg::Nil),"black")
     // to Auxg and Auxd with their weight,
    val ToAuxd    = post_condition("To"::prefid,"Auxr"::prefid,
                                  findWeightInPlace("Auxr"::prefid, Auxd::Nil),"black")
     //and post of To to Out
    val ToOut     = post_condition("To"::prefid,"Out"::prefid,1,"black")

    (PetriNet(In::Auxg::Auxd::Out::pn11.SetOfPlaces:::pn22.SetOfPlaces,
          To::pn11.SetOfTrans:::pn22.SetOfTrans,
          OutgTo::OutdTo::Pre1:::Pre2,
          ToAuxg::ToAuxd::ToOut::pn11.SetOfPost:::pn22.SetOfPost,
          pn1.SetOfPreSpaces:::pn2.SetOfPreSpaces,
          pn1.SetOfPostSpaces:::pn2.SetOfPostSpaces),
          valueSum(pc1,pc2)) // synthesis
}
```

Figure 11.44: The code for the construction of the Petri Net elements of a parallel composition of two agents (cont)

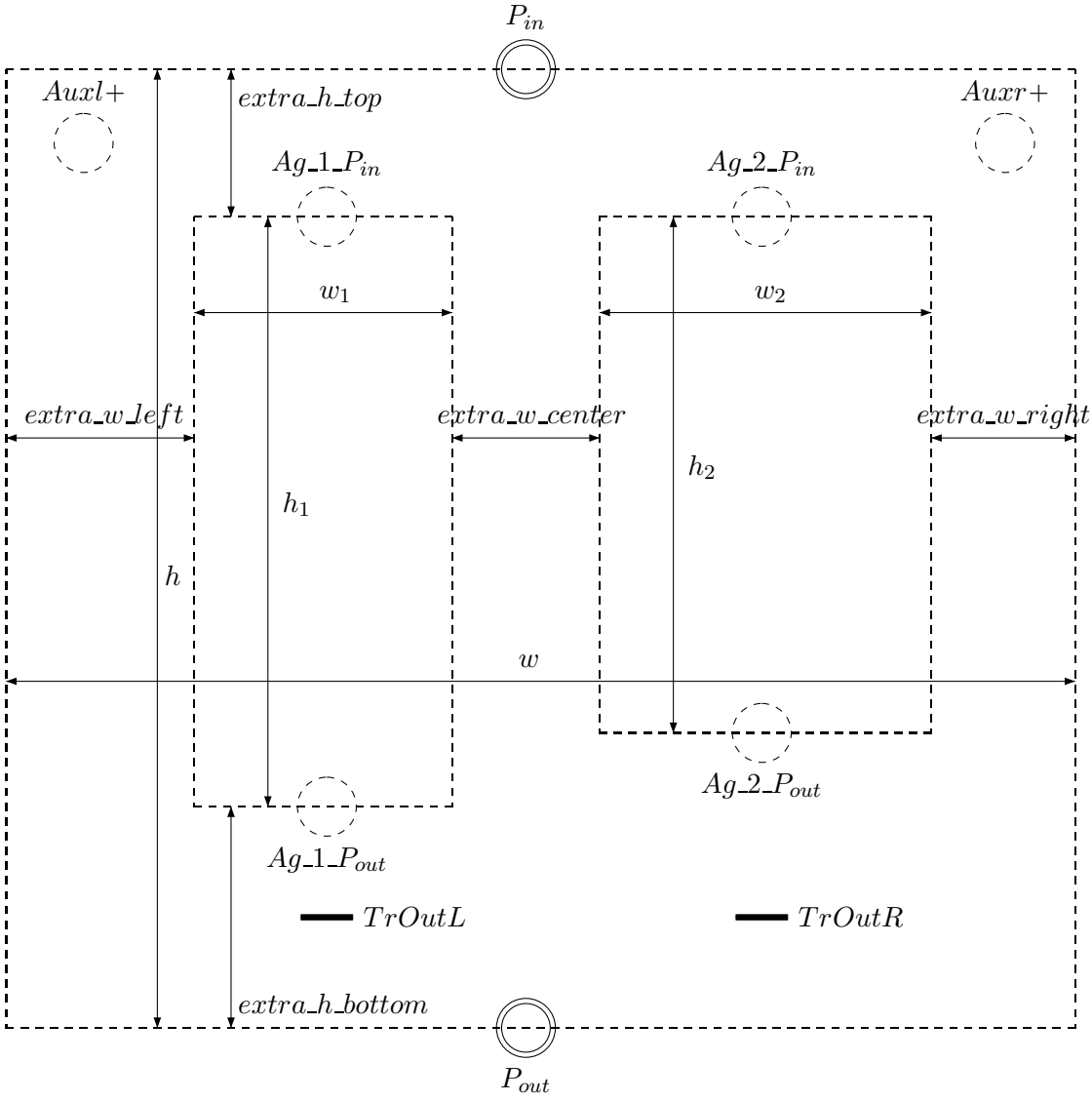


Figure 11.45: The schema of building of a complex agent based on two agents ag_1 and ag_2 for a choice composition

11.4.2.4 Choice composition

The construction of a choice compostion is done in a similar way to the one for the parallel composition. It starts with the definition of the X and Y coordinates of the center of the entry place of the choice agent, followed by the recursive calls to *convert2petriNet* for the construction of both sub-agents *ag_i* and *ag_ii*. The coordinates of the left upper corners of the rectangle are calculated with the constants *extra_w_center* and *extra_h_top*, of Figure 11.45. This figure is almost the same as the figure for the parallel composition. The only difference is in the presence of two transitions *TrOutL* and *TrOutR* for the propagation of the result of the execution of one of the two sub-agents to the final exit place. The presence of two transitions indicates that only one sub-agent has to be successful, for the choice agent to be successful.

```
val mean_X = posX + width/2
val mean_Y = posY + 10

var (pn1,pc1)  = convert2petriNet(ag_i,"1"::prefid,mean_X
             - extra_w_center/2 - ag_i.width,mean_Y + 2*extra_h_top/3)
var (pn2,pc2)  = convert2petriNet(ag_ii,"2"::prefid,mean_X
             + extra_w_center/2,mean_Y + 2*extra_h_top/3)
```

The entry *In* and exit *Out* places of the choice agent must be defined. Places of control, also called auxiliary places *Auxl+* and *Auxr+* have to be added to the choice agent. A "+"sign has been added to their name to distinguish them from their equivalent names for the parallel composition. Their role is also to control the execution of the sub-agents present in the composition. Nevertheless, and differing from the parallel composition, each of these places have the characteristic to be connected to both entries of the sub-agents. This permits to control, when an agent is selected among a choice, if the second agent has not been selected before. This implies to build pre conditions for the new auxiliary places with regard of the transitions related to the entry places for all the sub-agents, and also to build the post conditions corresponding to the return of these transitions, but only for the agent that is not on the same side of the control place. Figure 11.46 shows the code of the auxiliary places *Auxl+* and *Auxr+*.

The entry place receives a number of tokens equal to the maximum between the number of token of the Petri Net *pc1* associated with the first agent, and the number of tokens ot the Petri Net *pc2* associated with the second agent.

As in the parallel case, the auxiliary places receive the same number of tokens as in the entry place of their corresponding agent. This number is obtained via the identification of the

```
val In   = place(mean_X,mean_Y,radius,true,false,false,"",
                 valueMax(pc1,pc2),valueMax(pc1,pc2),"In"::prefid,"blue")

if (findInPlace(pn1.SetOfPlaces).length - In.idplace.length == 1) {
 Auxg = place(mean_X - width/2 + radius,mean_Y + extra_h_top - 2*radius,
           radius,false,false,true,"",findWeightInPlace("In"::"1"::prefid,
           pn1.SetOfPlaces),1,"Auxl+"::prefid,"black")
} else {
 Auxg = place(mean_X - width/2 + radius,mean_Y + extra_h_top - 2*radius,
           radius,false,false,true,"",findWeightInPlace("In"::"1"::"1"::prefid,
           pn1.SetOfPlaces),1,"Auxl+"::prefid,"black")
}
if (findInPlace(pn2.SetOfPlaces).length - In.idplace.length == 1) {
 Auxd = place(mean_X + width/2 - radius,mean_Y + extra_h_top - 2*radius,radius,
           false,false,true,"",findWeightInPlace("In"::"2"::prefid,
           pn2.SetOfPlaces),1,"Auxr+"::prefid,"black")
} else {
 Auxd = place(mean_X + width/2 - radius,mean_Y + extra_h_top - 2*radius,radius,
           false,false,true,"",findWeightInPlace("In"::"1"::"2"::prefid,
           pn2.SetOfPlaces),1,"Auxr+"::prefid,"black")
}

val Out  = place(mean_X,mean_Y + height,radius,false,true,false,"",0,1,
           "Out"::prefid,"black")
```

Figure 11.46: The code for the construction of the auxiliary places in case of a choice composition

place, that can be different in case of a sequential composition. This possibility is discussed by comparison of the length of the id list of the entry place of the choice agent, with the length of the id list of the entry place of the sub-agent.

Based on the collected pre conditions of the entry places *PrePn1In* and *PrePn2In*, all the pre conditions of the auxiliary places are constructed, for both sub-agents *ag_i* and *ag_ii*.

```
PrePn1Auxi1 = giveAll("Auxl+"::prefid, PrePn1In, 1, "black")
PrePn2Auxi1 = giveAll("Auxl+"::prefid, PrePn2In,
   findWeightInPlace("Auxl+"::prefid, Auxg::Nil), "black")
PrePn1Auxi2 = giveAll("Auxr+"::prefid, PrePn1In,
           findWeightInPlace("Auxr+"::prefid, Auxd::Nil), "black")
PrePn2Auxi2 = giveAll("Auxr+"::prefid, PrePn2In, 1, "black")
```

The post conditions of both control places with regard to the agent of the opposite place

are defined within the following code :

```
PostPn2Aux1 = newPostAux("Auxl+"::prefid, PrePn2In,
           findWeightInPlace("Auxl+"::prefid, Auxg::Nil))
PostPn1Aux2 = newPostAux("Auxr+"::prefid, PrePn1In,
           findWeightInPlace("Auxr+"::prefid, Auxd::Nil))
```

In this code the function *newPostAux* defines the post conditions of the place *Auxl+* with regard to the transitions of the agent constructed on the right part of the drawing. These transitions are obtained with the list of *prePn2In* of the pre conditions related to the entry place of the second agent. The same is done for the place *Auxr+*, with regard of the agent constructed on the right of the drawing.

The next step consists in removing the entry places of the sub-agents, and to replace them with the entry place of the choice agent. This operation implies also to adapt the pre conditions that concerns the entry places of the sub-agent. In these preconditions, the referenced place must be replaced by the entry place of the choice agent. The following code is responsible for these operations :

```
if(pc1 > pc2) {
 pn11 = removeInAndReplace("In"::prefid,1,pn1)
 pn22 = removeInAndReplace("In"::prefid,pc1,pn2)
} else {
  if(pc2 > pc1) {
   pn11 = removeInAndReplace("In"::prefid,pc2,pn1)
   pn22 = removeInAndReplace("In"::prefid,1,pn2)
} else {
   pn11 = removeInAndReplace("In"::prefid,1,pn1)
   pn22 = removeInAndReplace("In"::prefid,1,pn2)
   }
}
```

The code discusses the differences between the two sub-agents, essentially when the choice is constructed between a choice sub-agent and a parallel sub-agent. The case where the first agent needs a number of tokens bigger than the second one is exemplified by the situation of a first sub-agent with a parallel structure, and a second agent with a choice structure. Then every pre condition between the choice entry place and the transitions of the first agent will consume only one token, as every member of the parallel construction must be fired. This is indicated by 1 in the call to the *removeInAndReplace* procedure. For the second agent, its choice structure means that only one of its member will be fired. It needs then to consume

all the tokens of the choice entry place, which is indicated by *pc1* in the *removeInAndReplace* procedure. The symetric situation is described in the second part of the discussion. The last discussion describes the situation where both sub-agents have the same structure.

The last elements to be constructed are the two transitions *Tog* and *Tod* that transfer the result of the chosen agent to the final exit place. Their coordinates - X and Y of the upper left corner, and the width and height of the rectangle - are calculated with the constants *radius*, *extra_w_center* and *extra_h_bottom*. The code is the following :

```
val Tog = transition(mean_X - extra_w_center/2 - ag_i.width/2 -
             radius,mean_Y + height - 2*extra_h_bottom/3,2*radius,radius/3,
             "Tol","",0,"Tol"::prefid,"black")
val Tod = transition(mean_X + extra_w_center/2 + ag_ii.width/2- radius,
             mean_Y + height - 2*extra_h_bottom/3,2*radius,radius/3,"Tor",
             "",0,"Tor"::prefid,"black")
```

The pre and post conditions of these transitions must also be defined, inside the following code. In the case of the post conditions containing the auxiliary places, the number of tokens of the transition must be equal to the number of tokens inside the definition of the auxiliary places.

```
val OutgTog = pre_condition(findOutPlace(pn1.SetOfPlaces),
                                                "Tol"::prefid,1,"black")
val OutdTod = pre_condition(findOutPlace(pn2.SetOfPlaces),
                                                "Tor"::prefid,1,"black")
val TogAuxg = post_condition("Tol"::prefid,"Auxl+"::prefid,
                 findWeightInPlace("Auxl+"::prefid, Auxg::Nil),"black")
val TodAuxd = post_condition("Tor"::prefid,"Auxr+"::prefid,
                 findWeightInPlace("Auxr+"::prefid, Auxd::Nil),"black")
val TogOut  = post_condition("Tol"::prefid,"Out"::prefid,1,"black")
val TodOut  = post_condition("Tor"::prefid,"Out"::prefid,1,"black")
```

Finaly, all the constructed elements must be grouped in the structure of the Petri Net, as follows:

```
(PetriNet(In::Auxg::Auxd::Out::pn11.SetOfPlaces:::pn22.SetOfPlaces,
  Tog::Tod::pn1.SetOfTrans:::pn2.SetOfTrans,
  OutgTog::OutdTod::Pre1:::Pre2,
  Post2,
  pn1.SetOfPreSpaces:::pn2.SetOfPreSpaces,
  pn1.SetOfPostSpaces:::pn2.SetOfPostSpaces),
  valueMax(pc1,pc2)
```

The construction of the Petri Net is similar in its steps to the process followed for the parallel composition. The complete code for the construction of the Petri Net elements for a choice composition of two agents is listed in Figures 11.47 to 11.51.

11.4.3 Drawing Petri Net representations

The Petri Net being constructed, the next step is to draw a picture of it. This is achieved by constructing an xml file, wherein will be written all the instructions to draw in a svg format the produced Petri Net displayed in a browser.

The *densebachtopetri_ xml.scala* scala file defines a class *pn2xml* that is composed of a set of internal methods, on the one hand, and of one main method called *convertPn2Svg*, on the other hand. The latter is responsible for writing the xml file that produces the svg picture. The set of internal methods is not discussed in the text. It is composed of two subsets. The first one is composed of methods that write in the xml file the instruction for drawing the specific components of the Petri Net. The second one is composed of tool methods for the retrieval of useful pieces of information from the data structures of the Petri Net. The reader will find their code in appendix (see section H.1 in chapter H).

The main method *convertPn2Svg* has two arguments : the Petri Net produced by the *convert2petriNet* method, and a mapping that proposes a serialized list of the firable transitions in agreement with the state of the Petri Net at a given moment. The first action *openSvgFile* of this method is to open a file in writing. The second action *beginSVG* is to write inside this file a header grouping different pieces of information such as the xml version, the stylesheet, the type of encoding, the maximum dimensions (in pixels) of the drawing. A second part of the header defines the head of the arrow used to represent the pre and post conditions.

```
def convertPn2Svg(pn : PetriNet, mp : Map[Int, List[String]]) {
 openSvgFile()
 beginSVG()
 ...
```

To represent the Petri Net, four components need to be drawn : the places of the Petri Net, including the places and the anti places representing the tokens in the token space, the transitions, the pre conditions of a transition, represented by an arrow leaving a place and pointing to the transition, and the post conditions represented by an arrow leaving the transition and pointing to a place.

```
case dbach_ast_agent("+",ag_i,ag_ii,pc,width,height) => {
    // choice of agents ag_i and ag_ii to Petri Net
 /*
 build places In, Auxg+, Auxd+ and Out; (the "+" distincts auxiliary
   places created for a choice agent, from those for a parallel agent,
   as they appear in pre og ag_i and ag_ii)
 build transition Tog and Tod
 remove In of ag_i and ag_ii and replace them by In (of choice agent);
 modify related pre (of previous action);
 build pre of Auxg+ to transitions of ag_i AND ag_ii;
 build pre of Auxd+ to transitions of ag_i AND ag_ii;
 build the post of Tog to Auxg+, and Out
 build the post of Tod to Auxd+, and Out
 */

 val mean_X = posX + width/2
 val mean_Y = posY + 10

 var (pn1,pc1)  = convert2petriNet(ag_i,"1"::prefid,mean_X -
              extra_w_center/2 - ag_i.width,mean_Y + 2*extra_h_top/3)
 var (pn2,pc2)  = convert2petriNet(ag_ii,"2"::prefid,mean_X
             + extra_w_center/2,mean_Y + 2*extra_h_top/3)

   // number of tokens of In = summation of those of pc1 and pc2
 val In    = place(mean_X,mean_Y,radius,true,false,false,"",
              valueMax(pc1,pc2),valueMax(pc1,pc2),"In"::prefid,"blue")
 var Auxg = place(0,0,0,false,false,false,"",0,0,Nil,"")
 var Auxd = place(0,0,0,false,false,false,"",0,0,Nil,"")

 // take into account the sequentiality of two agents; the number of
 // token is the number of token of In place

 if (findInPlace(pn1.SetOfPlaces).length - In.idplace.length == 1) {
  Auxg = place(mean_X - width/2 + radius,mean_Y + extra_h_top - 2*radius,radius,
    false,false,true,"",findWeightInPlace("In"::"1"::prefid, pn1.SetOfPlaces),
    1,"Auxl+"::prefid,"black")
 } else {
  Auxg = place(mean_X - width/2 + radius,mean_Y + extra_h_top - 2*radius,radius,
    false,false,true,"",findWeightInPlace("In"::"1"::"1"::prefid, pn1.SetOfPlaces),
    1,"Auxl+"::prefid,"black")
 }
 if (findInPlace(pn2.SetOfPlaces).length - In.idplace.length == 1) {
  Auxd = place(mean_X + width/2 - radius,mean_Y + extra_h_top - 2*radius,radius,
    false,false,true,"",findWeightInPlace("In"::"2"::prefid, pn2.SetOfPlaces),1,
    "Auxr+"::prefid,"black")
 } else {
  Auxd = place(mean_X + width/2 - radius,mean_Y + extra_h_top - 2*radius,radius,
    false,false,true,"",findWeightInPlace("In"::"1"::"2"::prefid, pn2.SetOfPlaces),
    1,"Auxr+"::prefid,"black")
 }
```

Figure 11.47: The code for the construction of the Petri Net elements of a choice composition of two agents

```
val Out  = place(mean_X, mean_Y + height, radius, false, true, false, "", 0, 1,
                                                                  "Out" :: prefid, "black")

 var PrePn1Auxi1     : List[pre_condition]  = Nil
 var PrePn1Auxi2     : List[pre_condition]  = Nil
 var PrePn2Auxi1     : List[pre_condition]  = Nil
 var PrePn2Auxi2     : List[pre_condition]  = Nil
 var PrePn1Temp1     : List[List[String]]   = Nil
 var PrePn1Temp11    : List[List[String]]   = Nil
 var PrePn1In        : List[List[String]]   = Nil
 var PrePn2Temp2     : List[List[String]]   = Nil
 var PrePn2Temp22    : List[List[String]]   = Nil
 var PrePn2In        : List[List[String]]   = Nil
 var PrePn1Auxi11    : List[pre_condition]  = Nil
 var PrePn2Auxi22    : List[pre_condition]  = Nil
 var PostPn2Aux1     : List[post_condition] = Nil
 var PostPn1Aux2     : List[post_condition] = Nil
 var Post            : List[post_condition] = Nil
 var Post1           : List[post_condition] = Nil
 var Post2           : List[post_condition] = Nil
 var listNamePlaces  : List[List[String]]   = Nil
 var Pre1            : List[pre_condition]  = Nil
 var Pre2            : List[pre_condition]  = Nil
 var nbr             : Int = 1
```

Figure 11.48: The code for the construction of the Petri Net elements of a choice composition of two agents (cont)

```
if(findInPlace(pn1.SetOfPlaces).length > 3) { // if ag_i is a composition of agents
 PrePn1Temp1 = collectIdTransPre(pn1.SetOfPre, "In"::"1"::"1"::prefid)
} else {
 // collect the idtrans of Pre associated with the place IN::1 of pn1 (= of ag_i)
 PrePn1Temp1 = collectIdTransPre(pn1.SetOfPre, "In"::"1"::prefid)
}
if(findInPlace(pn2.SetOfPlaces).length > 3) { // if ag_ii is a composition of agents
 PrePn2Temp2 = collectIdTransPre(pn2.SetOfPre, "In"::"1"::"2"::prefid)
} else {
 PrePn2Temp2 = collectIdTransPre(pn2.SetOfPre, "In"::"2"::prefid)
}

if(findInPlace(pn1.SetOfPlaces).length > 2) { // if ag_i is a composition of agents
 PrePn1Temp11 = collectIdTransPre(pn1.SetOfPre, "In"::"1"::"1"::prefid)
} else {
 // collect the idtrans of Pre associated with the place IN::1 of pn1 (= of ag_i)
 PrePn1Temp11 = collectIdTransPre(pn1.SetOfPre, "In"::"1"::prefid)
}
if(findInPlace(pn2.SetOfPlaces).length > 2) { // if ag_ii is a composition of agents
 PrePn2Temp22 = collectIdTransPre(pn2.SetOfPre, "In"::"1"::"2"::prefid)
} else {
 PrePn2Temp22 = collectIdTransPre(pn2.SetOfPre, "In"::"2"::prefid)
}

if(PrePn1Temp1 == Nil) {
 PrePn1In = PrePn1Temp11
} else {
 PrePn1In = PrePn1Temp1:::PrePn1Temp11
}

if(PrePn2Temp2 == Nil) {
 PrePn2In = PrePn2Temp22
} else {
 PrePn2In = PrePn2Temp2:::PrePn2Temp22
}
```

Figure 11.49: The code for the construction of the Petri Net elements of a choice composition of two agents (cont)

```
// build all the pre of Auxg+ to the idtrans of ag_i
 // idem for Auxg+ to the idtrans of ag_ii
 // idem for Auxd+ to the idtrans of ag_i
 // idem for Auxd+ to the idtrans of ag_ii

 PrePn1Auxi1 = giveAll("Auxl+"::prefid, PrePn1In, 1, "black")
 PrePn2Auxi1 = giveAll("Auxl+"::prefid, PrePn2In,
                       findWeightInPlace("Auxl+"::prefid, Auxg::Nil), "black")
 PrePn1Auxi2 = giveAll("Auxr+"::prefid, PrePn1In,
                       findWeightInPlace("Auxr+"::prefid, Auxd::Nil), "black")
 PrePn2Auxi2 = giveAll("Auxr+"::prefid, PrePn2In, 1, "black")
 PostPn2Aux1 = newPostAux("Auxl+"::prefid, PrePn2In,
                       findWeightInPlace("Auxl+"::prefid, Auxg::Nil))
 PostPn1Aux2 = newPostAux("Auxr+"::prefid, PrePn1In,
                       findWeightInPlace("Auxr+"::prefid, Auxd::Nil))

 // remove In of ag_i and ag_ii, and replace them by In of choice agent in their pre

 var pn11 = PetriNet(Nil,Nil,Nil,Nil,Nil,Nil)
 var pn22 = PetriNet(Nil,Nil,Nil,Nil,Nil,Nil)
 if(pc1 > pc2) {
  pn11 = removeInAndReplace("In"::prefid,1,pn1)
  pn22 = removeInAndReplace("In"::prefid,pc1,pn2)
 } else {
  if(pc2 > pc1) {
   pn11 = removeInAndReplace("In"::prefid,pc2,pn1)
   pn22 = removeInAndReplace("In"::prefid,1,pn2)
  } else {
   pn11 = removeInAndReplace("In"::prefid,1,pn1)
   pn22 = removeInAndReplace("In"::prefid,1,pn2)
  }
 }

 var pnTotPlaces = pn11.SetOfPlaces:::pn22.SetOfPlaces

 listNamePlaces = collectIdPlaces(pnTotPlaces)
```

Figure 11.50: The code for the construction of the Petri Net elements of a choice composition of two agents (cont)

```
PrePn1Auxi11 = PrePn1Auxi1 ::: PrePn1Auxi2 ::: pn11.SetOfPre
PrePn2Auxi22 = PrePn2Auxi1 ::: PrePn2Auxi2 ::: pn22.SetOfPre
Pre1 = PrePn1Auxi11.distinct
Pre2 = PrePn2Auxi22.distinct

val Tog = transition(mean_X - extra_w_center/2 - ag_i.width/2 - radius,mean_Y
 + height - 2*extra_h_bottom/3,2*radius,radius/3,"Tol","",0,"Tol"::prefid,"black")
val Tod = transition(mean_X + extra_w_center/2 + ag_ii.width/2- radius,mean_Y
 + height - 2*extra_h_bottom/3,2*radius,radius/3,"Tor","",0,"Tor"::prefid,"black")

   // build the pre of places Outg and Outd to transition To
val OutgTog = pre_condition(findOutPlace(pn1.SetOfPlaces),"Tol"::prefid,1,"black")
val OutdTod = pre_condition(findOutPlace(pn2.SetOfPlaces),"Tor"::prefid,1,"black")

val TogAuxg = post_condition("Tol"::prefid,"Auxl+"::prefid,
                             findWeightInPlace("Auxl+"::prefid, Auxg::Nil),"black")
val TodAuxd = post_condition("Tor"::prefid,"Auxr+"::prefid,
                             findWeightInPlace("Auxr+"::prefid, Auxd::Nil),"black")
val TogOut  = post_condition("Tol"::prefid,"Out"::prefid,1,"black")
val TodOut  = post_condition("Tor"::prefid,"Out"::prefid,1,"black")

Post = TogAuxg::TodAuxd::TogOut::TodOut::PostPn2Aux1 ::: PostPn1Aux2
                                                  ::: pn11.SetOfPost ::: pn22.SetOfPost
Post = Post.distinct

if(listNamePlaces.contains("Out"::"1"::"1"::Nil)) { //
 Post1 = replaceAddTransInPost("Tr"::"1"::"1"::Nil, "Tr"::"2"::"1"::Nil,
                                                    "Tr"::"1"::Nil, Post)
 Post2 = replaceAddTransInPost("Tr"::"2"::"2"::Nil, "Tr"::"1"::"2"::Nil,
                                                  "Tr"::"2"::Nil, Post1)

 (PetriNet(In::Auxg::Auxd::Out::pn11.SetOfPlaces ::: pn22.SetOfPlaces,
           Tog::Tod::pn1.SetOfTrans ::: pn2.SetOfTrans,
           OutgTog::OutdTod::Pre1 ::: Pre2,
           Post2,
           pn1.SetOfPreSpaces ::: pn2.SetOfPreSpaces,
           pn1.SetOfPostSpaces ::: pn2.SetOfPostSpaces),
           valueMax(pc1,pc2))
} else { //
 (PetriNet(In::Auxg::Auxd::Out::pn11.SetOfPlaces ::: pn22.SetOfPlaces,
           Tog::Tod::pn1.SetOfTrans ::: pn2.SetOfTrans,
           OutgTog::OutdTod::Pre1 ::: Pre2,
           Post,
           pn1.SetOfPreSpaces ::: pn2.SetOfPreSpaces,
           pn1.SetOfPostSpaces ::: pn2.SetOfPostSpaces),
           valueMax(pc1,pc2)) // synthesis
}
}
}
}
```

Figure 11.51: The code for the construction of the Petri Net elements of a choice composition of two agents (cont)

These four steps are depicted into four blocs of code. The first bloc is the one that takes in charge the representation of all the pre conditions. For every pre conditions of the Petri Net, the drawing is done depending on the name of the starting place. Each time the X and Y coordinates of the starting point, situated on the border of a place circle, and of the arrival point of the arrow, situated on the border of a transition rectangle, have to be provided. It is to be noted that the pre conditions between the places and anti places of the token space, and their associated transitions are not drawn, essentially in the interests of clarity of the drawing. The following code shows an example of the drawing of the pre condition between an auxiliary place *Auxl+* in a choice composition and a transition. Every coordinate is obtained with the use of methods - like i.e. *findCoordXOfPlace* that extracts them from the adequate sets of elements in the Petri Net structure :

```
pn.SetOfPre.foreach( elm => {
 (elm.idplace).head match {
  case "Auxl+" => {
     drawP2Tcourbe(findCoordXOfPlace(elm.idplace,pn.SetOfPlaces)+radius,
       findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)-radius/6,
       findCoordXOfTrans(elm.idtrans,pn.SetOfTrans) +
                findWidthOfTrans(elm.idtrans,pn.SetOfTrans)/2,
       findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)
       +(findCoordYOfTrans(elm.idtrans,pn.SetOfTrans)-
                findCoordYOfPlace(elm.idplace,pn.SetOfPlaces))/3,
       findCoordXOfTrans(elm.idtrans,pn.SetOfTrans) +
                findWidthOfTrans(elm.idtrans,pn.SetOfTrans)/2,
       findCoordYOfTrans(elm.idtrans,pn.SetOfTrans)-radius,"green")
 } ...
```

The second part of the code depicts the way to draw all the post conditions of the Petri Net. This is done in a way similar to the pre conditions. Essentially the methods need the X and Y coordinates of the starting and ending points of the arrow, now coming from the transition, and going to a place. The curvature orientation of the arrow line being dependant from the distance between those two points, a discussion makes precise the code to be applied in function of this parameter (arbitrarily fixed to a value of 300). Again the post conditions between the transitions and the place of the token space have not been drawn for the sake of clarity of the drawing.

```
pn.SetOfPost.foreach( elm => {
 (elm.idplace).head match {
  case "Auxl" => {
```

```
   // if relative distance is bigger than 300
   if((findCoordYOfTrans(elm.idtrans,pn.SetOfTrans)-
                findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)) > 300) {
                  drawT2Pcourbe(findCoordXOfTrans(elm.idtrans,pn.SetOfTrans),
                  findCoordYOfTrans(elm.idtrans,pn.SetOfTrans) +
                              findHeightOfTrans(elm.idtrans,pn.SetOfTrans)/2,
                  findCoordXOfPlace(elm.idplace,pn.SetOfPlaces),
                  findCoordYOfTrans(elm.idtrans,pn.SetOfTrans),
                  findCoordXOfPlace(elm.idplace,pn.SetOfPlaces),
                  findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)+2*radius,"green")
 } else {
      drawT2Pcourbe(findCoordXOfTrans(elm.idtrans,pn.SetOfTrans)+
                                  findWidthOfTrans(elm.idtrans,pn.SetOfTrans),
      findCoordYOfTrans(elm.idtrans,pn.SetOfTrans),
      findCoordXOfTrans(elm.idtrans,pn.SetOfTrans),
      findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)-9*radius,
      findCoordXOfPlace(elm.idplace,pn.SetOfPlaces)+2*radius,
      findCoordYOfPlace(elm.idplace,pn.SetOfPlaces),"green")
 }
} ...
```

The third part of the code draws the rectangle of all the transitions, with their associated name. The drawing of the rectangle is made by the function *drawTransition*, based on the X and Y coordinates of the upper left corner of the rectangle, its width and height. A transition that is part of the mapping of the firable one is drawn in red. Otherwise it is drawn in black. Drawing in red is achieved by a function called *drawTransitionColour*. The code makes also the distinction between the transitions that are related to a Dense Bach primitive, from those that are not. In the first case, a text is drawn by the function *drawText* near the rectangle, with the name of the primitive, the name of the token and its density. In the second case, only the name of the transition is drawn by a function called *drawTransName* near its rectangle.

```
pn.SetOfTrans.foreach (elm => {
        // if NOT belongToMap, draw in black
 if(belongToMap(mp,elm.idtrans)._2 == Nil) {
  if(elm.token == "") {
   drawTransition(elm.tx,elm.ty,elm.width,elm.height)
   drawTransName(elm.tx + 18,elm.ty,elm.width,elm.height,elm.name)
  } else {
   drawTransition(elm.tx,elm.ty,elm.width,elm.height)
   drawText(elm.tx + 5,elm.ty,elm.width,elm.height,elm.name,elm.token,
                                                          elm.density)
  }
```

```
 } else { // else draw in colour
  if(elm.token == "") {
   drawTransitionColour(elm.tx,elm.ty,elm.width,elm.height)
   drawTransName(elm.tx + 18,elm.ty,elm.width,elm.height,elm.name)
  } else {
   drawTransitionColour(elm.tx,elm.ty,elm.width,elm.height)
   drawText(elm.tx + 5,elm.ty,elm.width,elm.height,elm.name,elm.token,
                                                              elm.density)
  }
 }
})
```

Finally the last part of the code concerns the representation of the places. This is made by the function *drawPlace*, based on the X and Y coordinates of the centre of the circle representing the place, and its radius. A distinction is done between the places that are specific to the Petri Net, and those that are part of the token space. For the first case, their drawing color differs in function of their type : the entry and exit places of the global agent are in blue, the auxiliary places are in green, and by default any other place is in black. Regarding the token space, as every token is associated with a pair of place and anti-place, the first category is drawn in black, and the second one is drawn in red. These places are drawn in a specific zone on the picture, that is determined with regard to the position of the entry place of the global agent. Every pair of place and anti place related to a specific token is drawn within, preceded by the name of the concerned token, thanks to the function *drawTokenName*. This zone is fixed by the coordinates *xOfIn* and *yOfIn*. For every place, the instantaneous number of tokens present in the place is written at its very centre. Figures 11.52 and 11.53 show the code for the drawing of the places, following their type.

The code finishes with the procedure *endSVG()* that writes the end xml marker in the file, and with *close_file()* to close the file. The complete code of the *convertPn2Svg* method is given in annex. (see section H.2 in Chapter H)

11.4.4 Running Petri Nets representations

After the construction of the Petri Net and its drawing in svg format, the last step consists in animating it dynamically according to the firing of the transitions. This is the purpose of the third class called *runningPetriNet* defined in the file *runningPetri.scala*. This class receives only the Petri Net as argument. Based on the sets of places, of transitions and of pre and post_conditions, the main procedure of this class evaluates for every state of the Petri Net the

```
var xOfIn : Int = 7*coordXOfIn(pn.SetOfPlaces)/4
var yOfIn : Int = coordYOfIn(pn.SetOfPlaces) + 18*radius

pn.SetOfPlaces.foreach( elm => {
 (elm.idplace).head match {
  case "In" => {
   drawPlace(elm.cx,elm.cy+radius,elm.radius,"blue")
   drawNbrTok(elm.cx,elm.cy+radius,elm.nbrTokens)
  }
  case "Auxl" => {
   drawPlace(elm.cx,elm.cy,elm.radius,"green")
   drawNbrTok(elm.cx,elm.cy,elm.nbrTokens)
  }
  case "Auxr" => {
   drawPlace(elm.cx,elm.cy,elm.radius,"green")
   drawNbrTok(elm.cx,elm.cy,elm.nbrTokens)
  }
  case "Auxl+" => {
   drawPlace(elm.cx,elm.cy,elm.radius,"green")
   drawNbrTok(elm.cx,elm.cy,elm.nbrTokens)
  }
  case "Auxr+" => {
   drawPlace(elm.cx,elm.cy,elm.radius,"green")
   drawNbrTok(elm.cx,elm.cy,elm.nbrTokens)
  } ...
```

Figure 11.52: The code for the drawing of the places.

.

```
  case "Out" => {
        // final out of the Petri Net in blue
   if ((elm.idplace).tail == Nil) {
    drawPlace(elm.cx,elm.cy,elm.radius,"blue")
    drawNbrTok(elm.cx,elm.cy,elm.nbrTokens)
   } else { // internal out of the Petri Net in black
drawPlace(elm.cx,elm.cy,elm.radius,"black")
drawNbrTok(elm.cx,elm.cy,elm.nbrTokens)
   }
  }
  case "ns" => {
   if (elm.name != "") {
drawTokenName(xOfIn + 3*radius/2,yOfIn,elm.name)
drawPlace(xOfIn + 7*radius/2,yOfIn,radius,"black")
drawNbrTok(xOfIn + 7*radius/2,yOfIn,elm.nbrTokens)
   }
  }
  case "as" => {
   if(elm.name != "") {
drawPlace(xOfIn + 13*radius/2, yOfIn,radius,"red")
drawNbrTok(xOfIn + 13*radius/2,yOfIn,elm.nbrTokens)
yOfIn += 3*radius
   }
  }
 }
})
```

Figure 11.53: The code for the drawing of the places (cont).

.

list of firable transitions. One of them being fired, the new state is displayed, and a new list of firable transitions is established. This process continues up to the moment where no more transition are firable, due to a deadlock or to the fact that the exit place of the Petri Net as been reached successfully.

As in the previous classes, the code is divided between a list of auxiliary tool functions, on the one hand, and three main functions, on the other hand. The tool procedures will not be described, and are available in annex (see section I.1 in Chapter I). The three essential functions are called *modifyInPetriNet*, *constructMapOfFirableTrans* and *execute*. Based on the modification of the Petri Net structure provided by *modifyInPetriNet*, and on the new list of firable transitions produced by *constructMapOfFirableTrans*, the *execute* procedure drives the process of renewing the Petri Net after each firing of a transition.

The first function called *modifyInPetriNet* has two arguments : the Petri Net *pn* and the identifier *idTrans* of a transition. For a definite transition name *idTrans*, the procedure encounters two possibilities: either the transitions are related to one of the four Dense Bach primitives, or not. In the first case the function creates the four lists *lTransPre*, *lTransPost*, *lTransPreSpaces* and *lTransPostSpaces* of pre and post_conditions valuable for the Petri Net as for the token space and executes them. This execution consists in substracting, thanks to the function *substractInPlace*, a required number of tokens in a place, and adding them, thanks to the function *addInPlace*, to a place. The targeted places are dictated by the definitions of the transitions. The required numbers of tokens for substraction and addition, are provided by the weights present in the pre or post_conditions. In the second case, i.e. for a transition with no relation with the places of the token space, only the lists *lTransPre* and *lTransPost* of pre and post_conditions are constructed, the principle of the calculation being the same. In both cases after the calculation, the new produced set of places *setAux* is added in the Petri Net, while the mapping of firable transitions is reset to an empty map. The result of this method is a new Petri Net, adapted to the modifications related to the transition.

Figures 11.54 and 11.55 present the pieces of code of the *modifyinPetriNet* function. Figure 11.54 shows the construction of the lists of pre and post conditions of transitions representing a primitive. In consequence, the lists of pre and post conditions concern the places of the Petri Net, as well as those related to the token space. After the construction of the lists specific to a primitive, the pre and post conditions are executed with the functions *addInPlace* and *substractInPlace*. They modify the content of the concerned places. Finally the Petri Net is adpated to the new representation. Figure 11.55 does the same operations but for transitions

not related to the token space.

```
 case _ => { // transitions concerned by Petri Net only
  lTransPre = subListPre(idTrans,pn.SetOfPre)
  lTransPost = subListPost(idTrans,pn.SetOfPost)

  lTransPre.foreach( x => {
   setAux = substractInPlace(x.idplace,setAux,x.weight)
  })

  lTransPost.foreach( x => {
   setAux = addInPlace(x.idplace,setAux,x.weight)
  })

  pn2Svg.convertPn2Svg(PetriNet(setAux,pn.SetOfTrans,pn.SetOfPre,
     pn.SetOfPost,pn.SetOfPreSpaces,pn.SetOfPostSpaces),Map())
  return PetriNet(setAux,pn.SetOfTrans,pn.SetOfPre,pn.SetOfPost,
        pn.SetOfPreSpaces,pn.SetOfPostSpaces)
 }
}
```

Figure 11.55: The code for the construction of the lists of pre and post conditions of transitions that are not primitives.

The second function *constructMapOfFirableTrans* has four arguments : a list of transitions *lTrans*, a list of places *lPlace*, and two lists *lPre* and *lPreSpaces* of pre_conditions, one valid for the Petri Net and the other for the token space. This function checks if every transition is firable or not. It then returns a mapping of all the firable transitions. For the transitions dedicated to a Dense Bach primitive, the checking is done with respect to the places of the Petri Net, thanks to the function *firablePre*, and the places of the token space, thanks to the function *firablePreSpaces*. For all the others, the check is performed with regard to the places of the Petri Net only, and with the use of function *firablePre*. In both cases every transition that is declared firable is then added to a mapping, finally returned to the user for a new submission. Figure 11.56 shows the code of function *constructMapOfFirableTrans*. Within the list of transitions *lTrans*, the function considers two cases. For those related to a Dense Bach primitive it checks if their associated pre conditions – those related to the Petri Net as well as those related to the token space – are firable. The function does the same for the transtions that are internal to the Petri Net, with no connection with the token space. At the end of the process, the function returns

```
def modifyInPetriNet(idTrans : List[String], pn : PetriNet) :
                                                    PetriNet = {
...
    // following type of transition
getNameInTrans(idTrans,pn.SetOfTrans) match {
          // transitions related to token space
 case "tell" | "get" | "ask" | "nask" => {
          // collect all pre of specific idTrans
  lTransPre = subListPre(idTrans,pn.SetOfPre)
  lTransPreSpaces = subListPreSpaces(idTrans,pn.SetOfPreSpaces)
  lTransPost = subListPost(idTrans,pn.SetOfPost)
  lTransPostSpaces = subListPostSpaces(idTrans,pn.SetOfPostSpaces)

  lTransPre.foreach( x => { // for every pre ...
          // substract weight of place idplace
   setAux = substractInPlace(x.idplace,setAux,x.weight)
  })

  lTransPost.foreach( x => { // for every post ...
         // add weight of place idplace
   setAux = addInPlace(x.idplace,setAux,x.weight)
  })
  lTransPreSpaces.foreach( x => { // for every pre of token space ...
         // substract weight
   setAux = substractInPlace(x.idplace,setAux,x.weight)
  })

  lTransPostSpaces.foreach( x => { // for every post of token space ...
        // add weight of place idplace
   setAux =  addInPlace(x.idplace,setAux,x.weight)
  })

     // convert adapted Petri Net to svg
  pn2Svg.convertPn2Svg(PetriNet(setAux,pn.SetOfTrans,pn.SetOfPre,
     pn.SetOfPost,pn.SetOfPreSpaces,pn.SetOfPostSpaces),Map())
  return PetriNet(setAux,pn.SetOfTrans,pn.SetOfPre,pn.SetOfPost,
        pn.SetOfPreSpaces,pn.SetOfPostSpaces)

 }
```

Figure 11.54: The code for the construction of the lists of pre and post conditions for the primitives, and their execution.

.

a map of all the firable transitions.

```
def constructMapOfFirableTrans(lTrans : List[transition],
                                                    lPlace : List[place],
  lPre : List[pre_condition],lPreSpaces : List[pre_condition]) :
                                          Map[Int, List[String]] = {
  ...
  lTrans.foreach(elm => { // for every transition of final Petri Net
   elm.name match {
case "tell" | "get" | "ask" | "nask" => { // evaluate if firable
 if(firablePre(elm.idtrans,lPre,lPlace) &&
              firablePreSpaces(elm.idtrans,lPreSpaces,lPlace)) {
  mapOfFirableTransL(i) = elm.idtrans
  firable = true
 }
}
case _ => { // for the others, evaluate if firable for Petri Net
 if(firablePre(elm.idtrans,lPre,lPlace)) {
  mapOfFirableTransL(i) = elm.idtrans
  firable = true
 }
}
   } // match
  if (firable) {
   i = i + 1
   firable = false
  }
 })
 return mapOfFirableTransL
}
```

Figure 11.56: The code of the *constructMapOfFirableTrans* function.

The third function called *execute* is the one that organizes the renewing of the Petri Net. It makes use of the two previous functions *modifyInPetriNet* and *constructMapOfFirableTrans*. Both are invoked in a loop initialized by *constructMapOfFirableTrans*. The user makes its choice in the proposed mapping. The function *modifyInPetriNet* then adapts the Petri Net. As a result, a new list of firable transitions is established, except if the last exit place of the Petri Net is reached, in which case the computation ends. This process continues also up to the moment where the mapping of firable transitions is empty. Figure 11.57 presents the code of the function *execute*. The argument *mapOfFirableTrans* receives the mapping containing all

```
def execute(pn : PetriNet) {

 var pnAux : PetriNet = pn
 mapOfFirableTrans = constructMapOfFirableTrans(pnAux.SetOfTrans,
                 pnAux.SetOfPlaces,pnAux.SetOfPre,pnAux.SetOfPreSpaces)
 pn2Svg.apply(pnAux,mapOfFirableTrans)

 while (! mapOfFirableTrans.isEmpty) {
  curentTrans = choiceFirableTrans(mapOfFirableTrans)
  println("Curent chosen transition : ")
  println(curentTrans)
  pnAux = modifyInPetriNet(curentTrans,pnAux)
   if(getNumbTokInPlace("Out"::Nil,pnAux.SetOfPlaces) == 0) {
mapOfFirableTrans = constructMapOfFirableTrans(pnAux.SetOfTrans,
             pnAux.SetOfPlaces,pnAux.SetOfPre,pnAux.SetOfPreSpaces)
pn2Svg.apply(pnAux,mapOfFirableTrans)
   } else {
mapOfFirableTrans = Map()
pn2Svg.apply(pnAux,mapOfFirableTrans)
   }
 }
}
```

Figure 11.57: The code of the *execute* function.

the firable transitions. A loop checking the emptyness of the map of firable transitions proposes to the user the firable one. The function *getNumbTokInPlace* takes the number of tokens inside the final *out* place. A zero means that this place has not yet been reached, and that a new mapping of firable transitions must be constructed. If the function returns a zero, the process ends.

The complete codes of the three procedures are listed in annex (see section I.2 in Chapter I).

11.4.5 Illustration on an example

In order to illustrate the functioning of the Petri Net, we describe hereafter the execution of an example. Lets us suppose we want to study the execution of the Dense Bach agent $(tell(u(2)) \ + \ tell(a(3)))\ ;\ (get(a(1)) \ || \ nask(u(3)))$. In the figures that follow, in place of putting black tokens in the places, we indicate explicitly their number.

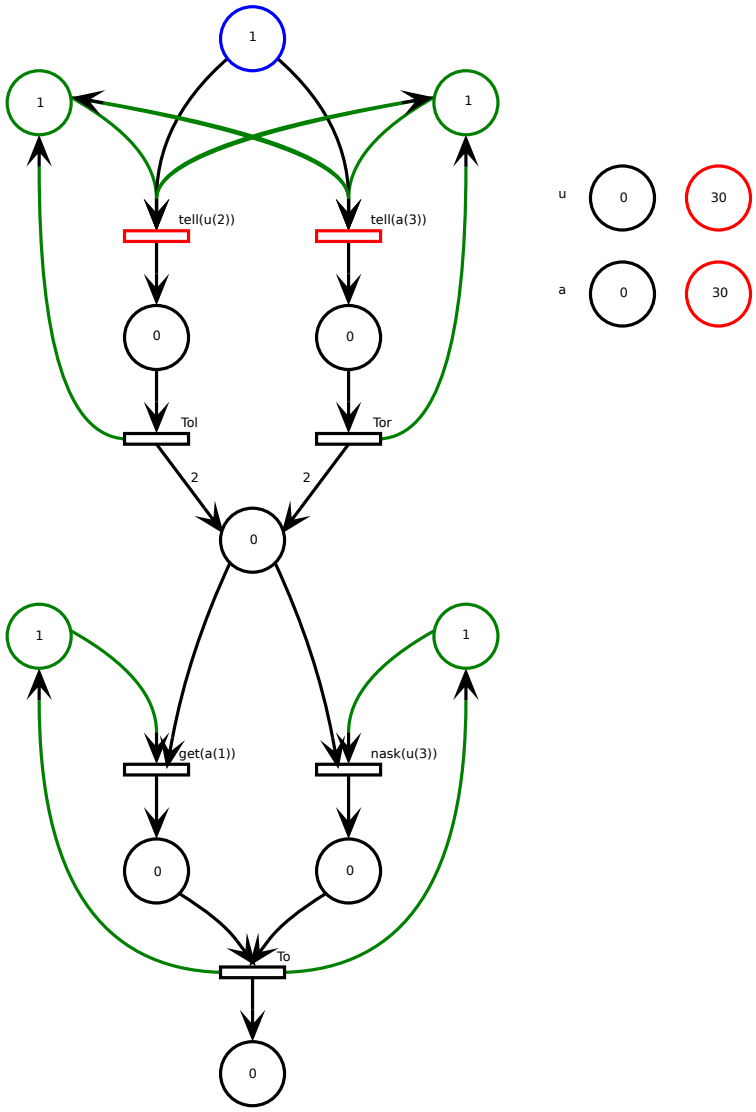


Figure 11.58: The initial state of the Petri Net associated with agent $(tell(u(2)) + tell(a(3)))\ ;\ (get(a(1)) \parallel nask(u(3)))$

.

The result of the parsing of this agent is analyzed by *convert2petriNet* to produce the Petri Net in its initial state. The transcription in svg made by *convertPn2Svg* produces a sequence of a choice structure, followed by a parallel one. The two auxiliary places of the choice and the parallel structures are set to 1. The upper blue place, corresponding to the entry place is also set to 1. There are two tokens manipulated by the agent : a and u. They are represented on the right part of the picture, and both initialized to 0 for the token space, and to 30 for the anti-space. This initial state is depicted in Figure 11.58. In all the figures that follow the tokens present in a place are always represented by a number written inside the places. The

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta http-equiv="refresh" content="5" />
  <title>Dense Bach in Petri</title>
</head>
<body>
  <center>
  <embed type="image/svg+xml" src="file:///home/ddarquen/thesisdda/DBtoPetri/file1xml" />
</center>

</body>
</html>
```

Figure 11.59: The code of the *execute* function.

four circles on the right part of the figure represent the tokens present in the token space. Every token is represented by its name. On the same line are drawn two circles, one in black and the second in red. The black one indicates the number of the tokens present in the token space. The red one indicates the number of tokens present in the anti-space. For the ease of presentation, the arrows between these token space places and the transitions of the primitives are not drawn.

The drawing of the Petri Net is loaded inside a web page, that is refreshed every five seconds. The html code for this refresh is presented in Figure 11.59.

To conduct the execution of the Petri Net, the map of the firable transitions produced by function *constructMapOfFirableTrans* explained in section 11.4.4 is presented to the user through a terminal. The following lines show the state of the terminal for the Net in its initial state of Figure 11.58. In them, the first element of each pair indicates the number to type in to indicate the user's choice. The second element provides the list in reverse order of the sub-component order. For instance, (2,1) denotes the second sub-agent of the first sub-agent of the considered agent. More specifically for the example under consideration, $(tell(u(2)) + tell(a(3)))$; $(get(a(1)) \,||\, nask(u(3)))$, the first sub-agent is $tell(u(2)) + tell(a(3))$ and its second sub-agent is $tell(a(3))$ so that (2,List(Tr, 2, 1)) refers to $tell(a(3))$.

```
List of the firable transitions:
```

```
((2,List(Tr, 2, 1)),
)
((1,List(Tr, 1, 1)),
)
Make your choice:
```

Looking to the composition of the first sub-agent, it proposes indeed a choice between two primitives: *tell(u(2))* and *tell(a(3))*. Both can be fired, but only one will be selected. Looking now to the parallel composition that follows sequentially, it is composed of the primitives *get(a(1))* and *nask(u(3))*. Both of them must be fired, but this depends on the result of the choice composition. If *tell(u(2))* is selected in the choice, two tokens will be added in the place u, and two will be retrieved in its corresponding anti-place. The firing of the *nask(u(3))* primitive will be succesful, as the number of u tokens present on the store does not exceed the limit of 3 requested by the *nask* primitive. Nevertheless the *get(a(1))* of primitive will not be succesful, if no token a is present on the token space. In order to make both *get(a(1))* and *nask(u(3))* firable within the parallel composition, *tell(a(3))* must be selected in place of *tell(u(2))*.

In the choice proposed to the user by the *constructMapOfFirableTrans* function, the number 2 for the *tell(a(3))* primitive is selected. Firing the *tell(a(3))* primitive puts 3 tokens in the place of a, and reduces the number 30 to 27 in the corresponding anti-place. The selection of this primitive means that the token present in the right auxiliary place must also been consumed, in order to lock the firing of the *tell(u(2))* primitive. The firing also consumes the token in the left auxiliary place, in order to check if the *tell(u(2))* has not been selected before. It produces and puts it back immediatley, as it produces one token in the exit place of the *tell(a(3))* primitive. This first evolution of the Petri Net is represented in Figure 11.60.

A new set of firable transitions is now established, that is shown in Figure 11.61.

```
Curent chosen transition :
List(Tr, 2, 1)
List of the firable transitions:

((1,List(Tor, 1)),
)
Make your choice:
```

Figure 11.61: The code of the *execute* function.

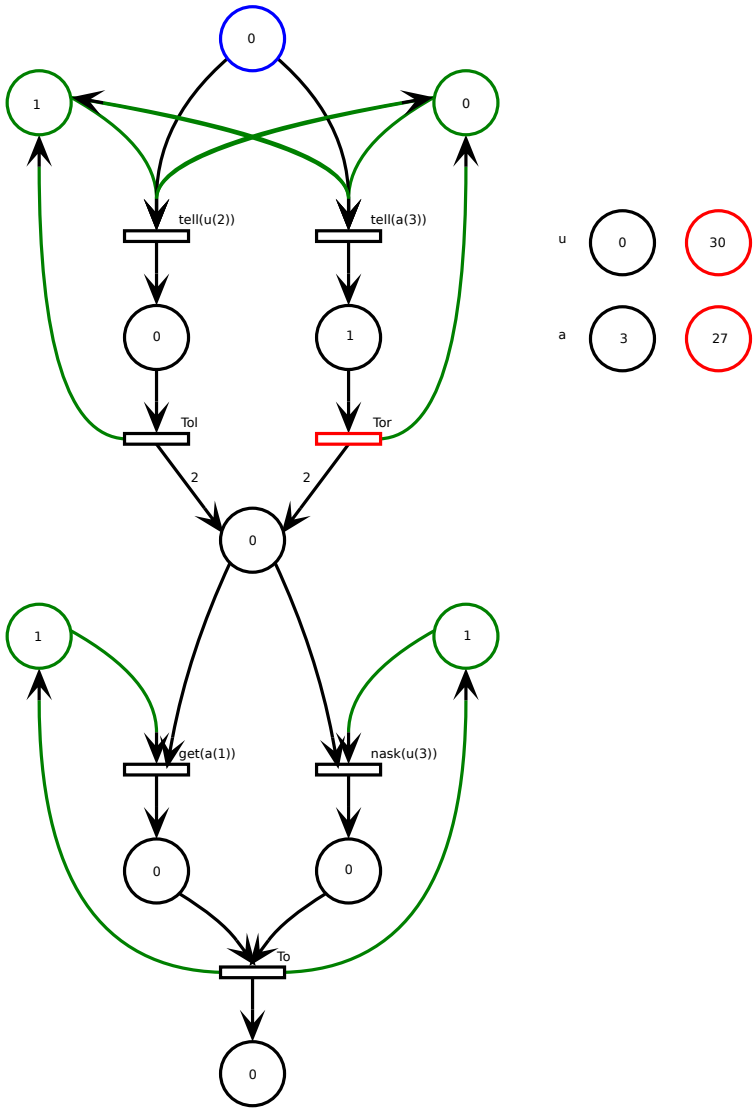


Figure 11.60: The result of the firing of *tell(a(3))* in the choice sub-agent

In order to conclude the execution of the choice, the right transition *Tor* has now to be fired. It consumes the token in the exit place of the *tell(a(3))* primitive, and performs two actions : the restore of the right auxiliary place to 1, and the supply of the entry place of the parallel composition following sequentially the choice. As both branches of the parallel composition must be fired, the number of tokens in the entry must be at least equal to 2, in place of 1. At this state, as a next step to perform, the user has now the choice between firing *get(a(1))* and *nask(u(3))*, both being firable, and both having to be fired. Figure 11.62 depicts the state of the Petri Net after the complete execution of the choice sub-agent.

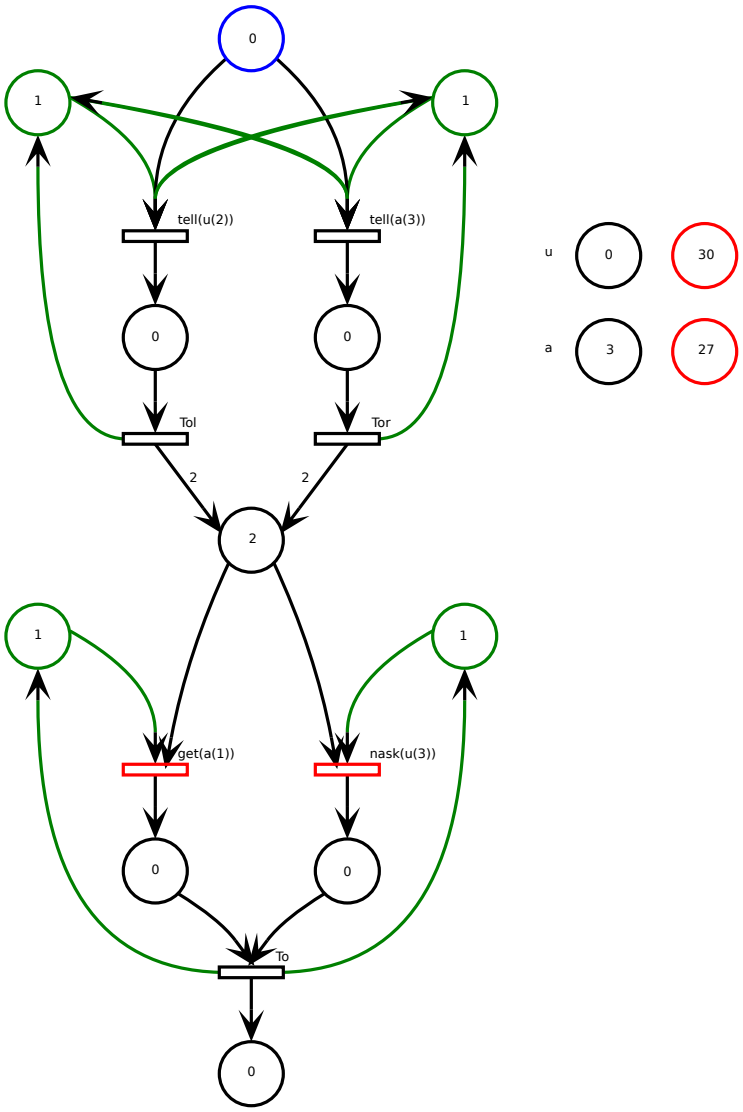


Figure 11.62: The result of the firing of the *Tor* transition concluding the choice sub-agent

A new set of firable transitions is again established, that is shown in Figure 11.63.

```
Curent chosen transition :
List(Tor, 1)
List of the firable transitions:

((2,List(Tr, 2, 2)),
)
((1,List(Tr, 1, 2)),
)
Make your choice:
```

Figure 11.63: The code of the *execute* function.

Let us suppose that the user chooses the *nask(u(3))* primitive. One on the tokens present in the entry place is consumed, as the token present in the right auxiliary place. This last action locks the possibility for the *nask(u(3))* primitive to be fired twice, by consuming the second token remaining in the entry place. The firing of the *nask(u(3))* primitive will not change the state of token u in the token space. Figure 11.64 depicts the state of the Petri Net after having fired the primitive *nask(u(3))* of the parallel composition.

In the new list of firable transitions, only the transition of the *get(a(1))* is available, as shown in Figure 11.65.

```
Curent chosen transition :
List(Tr, 2, 2)
List of the firable transitions:

((1,List(Tr, 1, 2)),
)
Make your choice:
```

Figure 11.65: The code of the *execute* function.

For the next step, the user has only the choice to fire the *get(a(1))*. It consumes the last token in the entry place, as the token present in the right auxiliary place. One token is produced and placed in the exit place of the primitive. Regarding the token space, the effect of the firing of the *get(a(1))* primitive is to take one unit in the black place corresponding to token a and to add it to the 27 present in the corresponding red anti-place. Figure 11.66 depicts the state of the Petri Net after the primitive *get(a(1))* of the parallel composition has been fired.

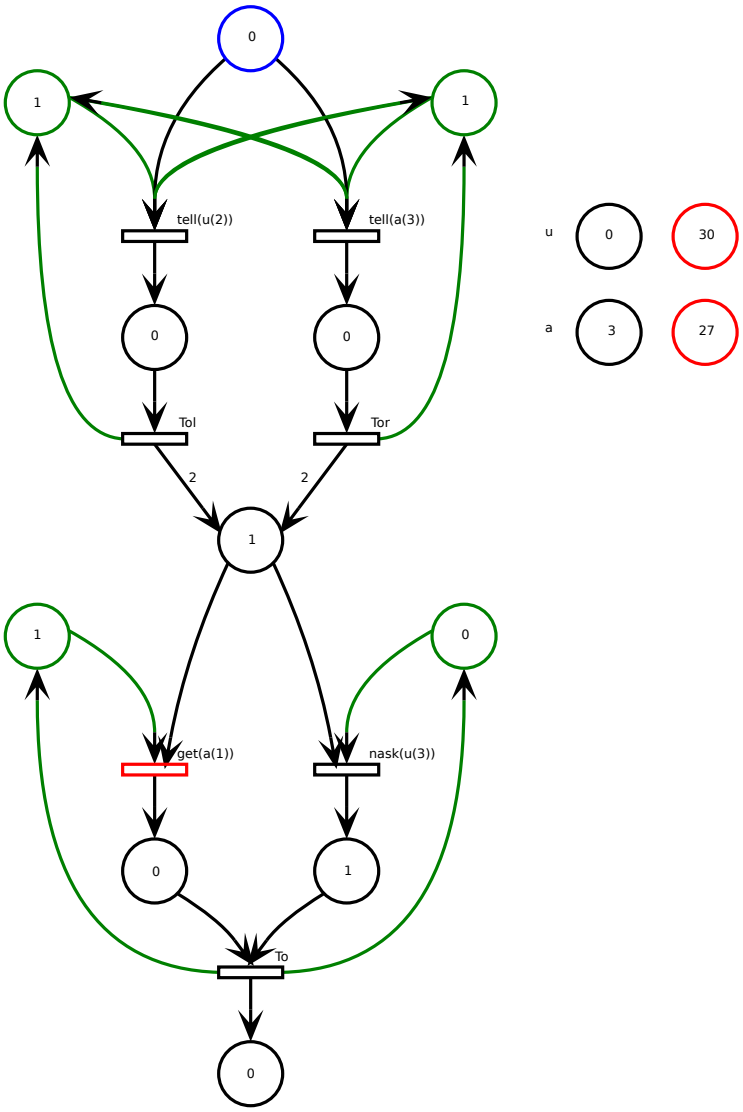


Figure 11.64: The result of the firing of the primitive *nask(u(3))* in the parallel sub-agent

Figure 11.67 shows the new selection of firable transitions.

```
Curent chosen transition :
List(Tr, 1, 2)
List of the firable transitions:

((1,List(To, 2)),
)
Make your choice:
```

Figure 11.67: The code of the *execute* function.

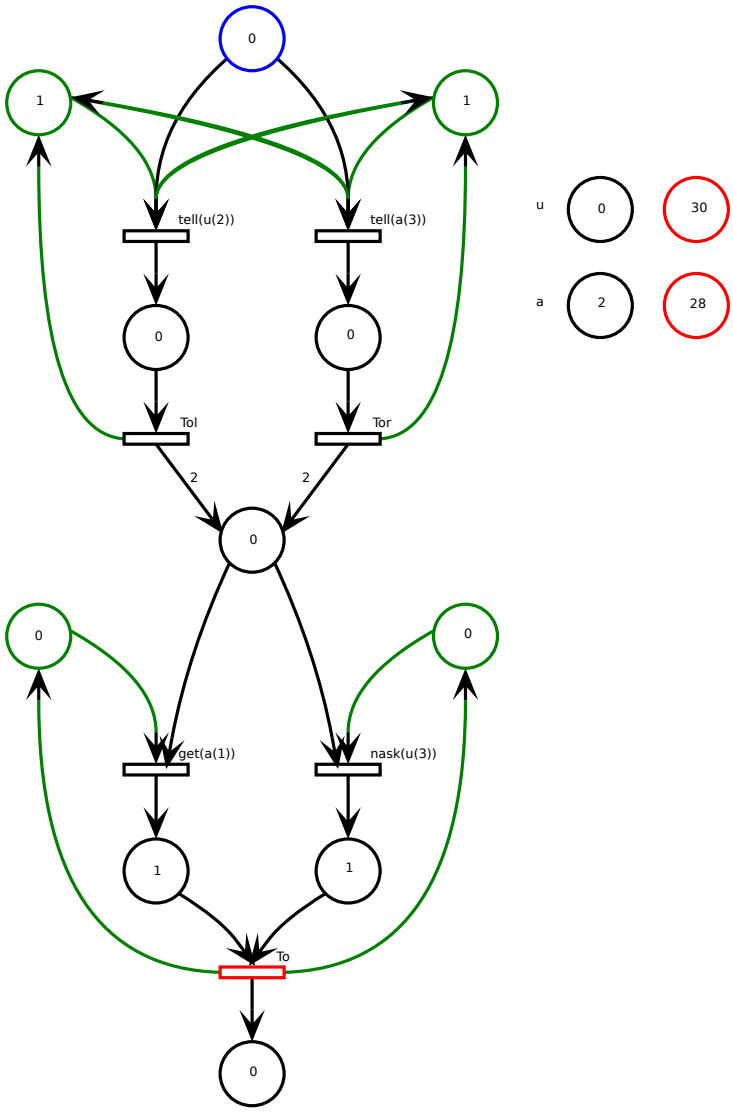


Figure 11.66: The result of the firing of the second primitive *get(a(1))* in the parallel sub-agent

As a final step, the last transition to be fired is *To*, that concludes the parallel composition. Consuming the tokens present in both exit places of the primitives *get(a(1))* and *nask(u(3))*, it restores the situation of the right and left auxiliary places, and puts one token in the final exit place. The state of the token space stays unchanged. The final result of the execution of the agent is then to have 2 tokens a in the store, and none for token u. The restore of the state of the auxiliary places is done and permits an iterative execution of agents, that is not considered in our thesis. Figure 11.68 depicts the final state of the Petri Net after the firing of the transition *To* of the parallel composition. The auxiliary places of the parallel composition are restored in their initial state.

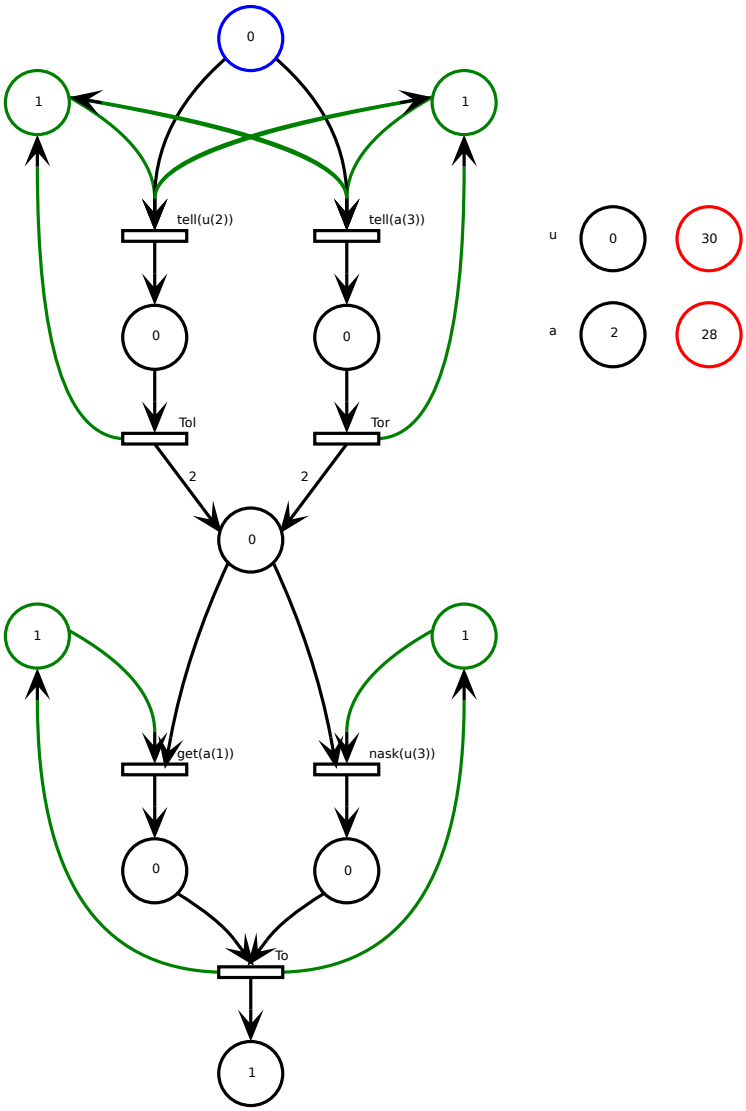


Figure 11.68: The result of the firing of the transition *To* in the parallel sub-agent, concluding the execution of the global agent

11.5 Conclusion

In this chapter we have developped an application in Scala that, using the result of the parsing of a Dense Bach agent, constructs its equivalent Petri net representation. The model for the representation of an agent is adapted from the concept of Open Petri Net. The Scala code for the transcription and construction of the Petri Net first constructs a Petri Net object, with four elements : a set of places, a set of transitions and two sets, and two sets for respectively the representation of the pre and post-conditions for the firing of the transitions. This object is extended by a token space, for the representation of the the tokens manipulated by the Dense Bach agents. Every token is associated with a duo of place anti-place for the representation of the actions of the primitives.

The second part of the code uses the created Petri net object and expresses it in a set of instructions for drawing the Net in a svg format. These instructions are encapsulated in xml instructions and written in xml file, in order to be executed in a browser.

Finally, the Petri Net is executed step by step, thanks to a selection of firable transitions. After each firing the representation of the Net is refreshed, and a new list of firable transitions is established, this up to the moment the agent is completely executed, or there is no more transitions to be fired.

Besides the intuitive modelling they provide, Petri Nets are popular for validating and verifying systems. The translation into Petri Nets we have proposed has been used in this chapter for simulation purposes. However, our translation provides also a means to perform deadlock detection, invariant analysis and, more generally, model checking of Dense Bach agents by reimplementing the classical algorithms developped for Petri Nets. Another way to reach this goal would be to generate xml format devoted to Petri Nets, like PNML, the Petri Net Markup Language, and use already developped tools like Tina. In these lines, it is worth noting that a translation in PNML follows similar lines to the one we have developped to generate svg graphs. Such developments are however left for future work.

Part IV

Conclusion

Chapter 12

Conclusion

It is commonly acknowledged that our life is ruled by information dynamically disseminated in different kinds of networks. Our current use of mobile devices particularly evidences this phenomenon. Mobile applications like Uber are widely spread and used to such non digital tasks as transport. In this context, it is of crucial importance to get usage feedbacks in order to guarantee not only satisfaction and quality of service but also security.

At the programming level, coordination languages have been proven very suited to code such applications thanks to the clear separation of concerns between computations and communication they embody as well as thanks to the time and space decoupling provided by the shared space used as communication medium. However coordination languages fail to address popularity and quality concerns that modern applications need to face.

Our thesis is that extending coordination languages to handle atomically finite numbers of tokens provides the necessary mechanisms. We have provided a support to our claim throughout this document by proposing different extensions of the Bach coordination language developped at the University of Namur, by studying the expressiveness of the resulting languages, by providing interperters and command line simulators, a graphical simulator as well as a translation to Petri Nets, that itself offers means to analyze programs written in our languages.

More concretely, restricting to tokens to better grasp the core concepts, a first extension, named Dense Bach, has consisted in associating a notion of density with the tokens manipulated by the Bach primitives. In order to treat several tokens at once, we have then introduced vectors of dense tokens in a new language called Vectorized Dense Bach and have shown how differents strategies of distributing a density among different tokens and of handling cardinalities can be derived therefrom.

Our expressiveness studies has allowed to compare our languages with two languages naturally related : on the one hand, the Bach language reduced to tokens, referred in the thesis as BachT and, on the other hand, a chemically inspired language, named MRT. Previous research by our advisors had already demonstrated the expressiveness relations between the sublanguages of BachT and of MRT. We have proven in our thesis that our languages fulfill these relations while being comprised in terms of expressiveness between the BachT and MRT languages, with Dense Bach a less expressive language than Vectorized Dense Bach.

We have developed interpreters and command line simulators for each language. The interpreters aims at allowing the reader to become familiar with the writing and the execution of programs. The command line simulators aim at introducing more flexibility by using threads to offer real concurrent execution and thus to allow to grasp suspension on conditions not fulfilled by the contents of the store, and this until other agents modify it to meet these conditions. As the interpreters and the command line simulators work in command line, we have developed a simulator to propose a more visual way of handling executions. Finally, in order to offer to users a tool to study the properties of programs we have developed a tool to transform Dense Bach agents into Petri Nets. This has been achieved through a newly notion of Open Petri Nets, named BD-open Petri Nets.

Our thesis opens research for future work. At the language level, we have concentrated on tokens instead of more structured tuples or ψ-terms (see [BJL06]) and have focussed on finite processes, thereby putting aside recursive definitions. Future work should address these topics and the consequences of their introduction on expressiveness, interperters/command line simulators and translation to Petri Nets.

Moreover, our implementations are based on Scala and have used classical data structures together with traditional locking mechanisms of the store. It would be interesting to analyze how much these implementation choices affect the performances of our interpreters and command line simulators by developping, for instance, command line simulators in C or by ruling the access to the store by partitioning techniques.

Finally, at the programming level, the graphical simulator naturally suggests to design representation techniques to illustrate computations. We believe that such aspects, typically neglected by computer scientists, are of a crucial importance for the dessimination of coordination languages and could open its utilization to other fields such that system biology.

Part V

Appendix

Appendix A

Appendix: Expressiveness of BachT and MRT

The two following sections present the main expressiveness results between the different sublanguages of the BachT family, and between the sublanguages of BachT and MRT, with their proofs.

A.1 Expressiveness relations between the BachT sublanguages

As most of the results are already established in [BJ03a, BJ98, BJ99], we subsequently make clear whether the proofs follow the main lines of [BJ03a, BJ98, BJ99] or whether they provide new reasonings.

A.1.1 Sublanguages

As a first result, by sublanguage inclusion (pattern 1), a number of modular embeddings are directly established.

Proposition 1. $\mathcal{L}_B(\ \psi) \leqslant \mathcal{L}_B(\ \chi)$, for any subsets of ψ, χ of primitives such that $\psi \subseteq \chi$.

A.1.2 Checking for presence and/or absence when adding tokens

Proposition 2. $\mathcal{L}_B(\text{tell}) < \mathcal{L}_B(\text{ask,tell})$

Proof. As in [BJ98], the proof takes advantage of proposition 1 to establish that $\mathcal{L}_B(\text{tell}) \leqslant \mathcal{L}_B(\text{ask,tell})$ and uses a contradiction technique to establish that $\mathcal{L}_B(\text{ask,tell}) \nleqslant \mathcal{L}_B(\text{tell})$. Indeed

considering the agent $ask(t)$ with $\mathcal{O}(ask(t)) = \{(\varnothing, \delta^-)\}$, we have a contradiction by property P3 of termination invariance, as any agent in $\mathcal{L}_B$(tell) has only successful computations. □

Proposition 3. $\mathcal{L}_B$(tell) $<$ $\mathcal{L}_B$(nask,tell)

Proof. As in [BJ98], the technique of the proof is analogous to the one of proposition 2. In particular the contradiction is established by considering the failing behaviour of the agent $tell(t)$; $nask(t)$, where any agent in $\mathcal{L}_B$(tell) has only successful computations. □

Proposition 4. $\mathcal{L}_B$(ask,tell) $\wr$ $\mathcal{L}_B$(nask,tell)

Proof. As in [BJ98], both components of the proof are established by contradiction.

(i) On the one hand, let us establish that $\mathcal{L}_B$(ask,tell) $\not\leqslant$ $\mathcal{L}_B$(nask,tell) by contradiction assuming a coder $\mathcal{C}$. For $A = tell(t)\,;\,ask(t)$, one has $\mathcal{O}(A) = \{(\{t\}, \delta^+)\}$. Hence, by P_3, $\mathcal{C}(A)$ succeeds whereas we shall establish that it has failing computations. Indeed, since $\mathcal{O}(ask(t)) = \{(\varnothing, \delta^-)\}$, any computation of $\mathcal{C}(ask(t))$ starting on the empty store fails. As $\mathcal{C}(ask(t))$ is composed of nask and tell primitives, this can only occur by having a nask primitive preceded by a tell primitive. As enriching the initial content of the store leads to the same result, any computation starting on any (arbitrary) store fails. As a consequence, even if $\mathcal{C}(tell(t))$ has a successful computation, this computation cannot be continued by a successful computation of $\mathcal{C}(ask(t))$. Consequently any computation of $\mathcal{C}(tell(t); ask(t))$ fails, which produces the announced contradiction.

(ii) On the other hand, $\mathcal{L}_B$(nask,tell) $\not\leqslant$ $\mathcal{L}_B$(ask,tell) is also established by contradiction. Assume a coder $\mathcal{C}$ and consider $A = tell(t)\,;\,nask(t)$. One has $\mathcal{O}(A) = \{(\{t\}, \delta^-)\}$. By P_3, the agent $\mathcal{C}(A)$ fails, whereas we shall establish that it has a successful computation. Indeed, since $\mathcal{O}(tell(t)) = \{(\{t\}, \delta^+)\}$, any computation of $\mathcal{C}(tell(t))$ starting on the empty store is successful. Similarly, it follows from $\mathcal{O}(nask(t)) = \{(\varnothing, \delta^+)\}$ that any computation of $\mathcal{C}(nask(t))$ starting on the empty store is successful. Consequently, so does any computation starting from any store, since $\mathcal{C}(nask(t))$ is composed of ask and tell primitives. Summing up, any (successful) computation of $\mathcal{C}(tell(t))$ starting on the empty store can be continued by a (successful) computation of $\mathcal{C}(nask(t))$, which leads to the announced contradiction. □

The reasoning used in the second part of proposition 4 will be used for proposition 8, for establishing that $\mathcal{L}_B$(nask,tell) $\not\leqslant$ $\mathcal{L}_B$(get,tell). Indeed, replacing ask by get in the $\mathcal{L}_B$(ask,tell) sublanguage does not change the fact that, with $\mathcal{O}(nask(t)) = \{(\varnothing, \delta^+)\}$, any computation of $\mathcal{C}(nask(t))$ is successful starting from any store. This then insures that any successful computation of $\mathcal{C}(tell(t))$ can be followed by a successful computation of $\mathcal{C}(nask(t))$, this leading to an obvious contradiction.

Proposition 5. $\mathcal{L}_B$(nask,tell) $< \mathcal{L}_B$(ask,nask,tell)

Proof. Both parts of the proof are established as in [BJ98]. **(i)** $\mathcal{L}_B$(nask,tell) $\leqslant$ $\mathcal{L}_B$(ask,nask,tell) results from proposition 1. **(ii)** $\mathcal{L}_B$(ask,nask,tell) $\not\leqslant$ $\mathcal{L}_B$(nask,tell) is established by pattern 3 on non embedding by transitivity, which leads to $\mathcal{L}_B$(ask,tell) $\leqslant$ $\mathcal{L}_B$(nask,tell), hence contradicting proposition 4. □

Proposition 6. $\mathcal{L}_B$(ask,tell) $< \mathcal{L}_B$(ask,nask,tell)

Proof. Both parts of the proof are established as in [BJ98]. **(i)** $\mathcal{L}_B$(ask,tell) $\leqslant \mathcal{L}_B$(ask,nask,tell) results from proposition 1. **(ii)** $\mathcal{L}_B$(ask,nask,tell) $\not\leqslant \mathcal{L}_B$(ask,tell) is established by pattern 3, which leads to $\mathcal{L}_B$(nask,tell) $\leqslant \mathcal{L}_B$(ask,tell), contradicting proposition 4. □

A.1.3 Retrieving tokens from the store

Proposition 7. $\mathcal{L}_B$(ask,tell) $< \mathcal{L}_B$(get,tell)

Proof. **(i)** As in [BJ98], the method of the proof invokes on the one hand a direct translation of the $ask(t)$ primitive in a $get(t)$; $tell(t)$ to establish $\mathcal{L}_B$(ask,tell) $\leqslant$ $\mathcal{L}_B$(get,tell). **(ii)** On the other hand and differently from [BJ98] $\mathcal{L}_B$(get,tell) $\not\leqslant$ $\mathcal{L}_B$(ask,tell) is established by a contradictory reasoning. Assume thus a coder $\mathcal{C}$ and consider $A = tell(t)$; $get(t)$. One has $\mathcal{O}(A) = \{(\varnothing, \delta^+)\}$. By P_2 and P_3, any computation of $\mathcal{O}(\mathcal{C}(tell(t))$; $\mathcal{C}(get(t))$ is thus successful. Such a computation is composed of a computation for $\mathcal{C}(tell(t))$ followed by a computation for $\mathcal{C}(get(t))$. As $\mathcal{C}(get(t))$ is composed of ask and tell primitives and since ask and tell primitives do not destroy elements, this latter computation can be repeated, which yields successful computations for $\mathcal{O}(\mathcal{C}(tell(t))$; $\mathcal{C}(get(t))$; $\mathcal{C}(get(t)))$. However, $\mathcal{O}(tell(t)$; $get(t)$; $get(t)) = \{(\varnothing, \delta^-)\}$, which leads to the contradiction. □

The reasoning used in the second part of proposition 7 can be adapted in order to be re-used in part (ii) of proposition 10, for establishing that $\mathcal{L}_B$(get,tell) $\not\leqslant$ $\mathcal{L}_B$(ask,nask,tell) and in part (ii) of proposition 27 of section A.2. The same agent $A = tell(t)$; $get(t)$ is now coded not only with primitives *tell* and *ask*, but also with *nask*. Nevertheless, the presence of the *nask* primitive can be dealt with by replacing the sequential composition of get(t) with himself, by a parallel composition, and by mimicking each step of $\mathcal{C}(get(t))$ in the computation of the other instance of $\mathcal{C}(get(t))$. This proof method is different from the one developped in [BJ98].

Proposition 8. $\mathcal{L}_B$(nask,tell) $\wr$ $\mathcal{L}_B$(get,tell)

Proof. The proof methods of both parts are the same as in [BJ98].**(i)** On the one hand, by using the same reasoning as in the second part of proposition 4, $\mathcal{L}_B$(nask,tell) $\not\leqslant$ $\mathcal{L}_B$(get,tell). **(ii)** On the other hand, by the pattern 3 of non embedding by transitivity, if $\mathcal{L}_B$(get,tell) $\leqslant$ $\mathcal{L}_B$(nask,tell), then $\mathcal{L}_B$(ask,tell) $\leqslant$ $\mathcal{L}_B$(nask,tell), which contradicts proposition 4. □

Proposition 9. $\mathcal{L}_B$(get,tell) = $\mathcal{L}_B$(ask,get,tell)

Proof. The proof methods of both parts are the same as in [BJ98]. **(i)** On the one hand $\mathcal{L}_B$(get,tell) $\leqslant$ $\mathcal{L}_B$(ask,get,tell) results from proposition 1. **(ii)** On the other hand, $\mathcal{L}_B$(ask,get,tell) $\leqslant$ $\mathcal{L}_B$(get,tell) is obtained by expressing each $ask(t)$ primitive as $get(t)$; $tell(t)$. □

Proposition 10. $\mathcal{L}_B$(ask,nask,tell) $\wr$ $\mathcal{L}_B$(get,tell)

Proof. **(i)** $\mathcal{L}_B$(ask,nask,tell) $\not\leqslant$ $\mathcal{L}_B$(get,tell) is established by the pattern 3 of non embedding by transitivity. Indeed if $\mathcal{L}_B$(ask,nask,tell) $\leqslant$ $\mathcal{L}_B$(nask,tell), then $\mathcal{L}_B$(ask,tell) $\leqslant$ $\mathcal{L}_B$(nask,tell), which contradicts proposition 4. **(ii)** $\mathcal{L}_B$(get,tell) $\not\leqslant$ $\mathcal{L}_B$(ask,nask,tell) is established by contradiction, similarly to the second part of proposition 7, by replacing the sequential composition of the two $get(t)$ primitives by a parallel one, in order to cope with the potential presence of nask primitives. □

A.1.4 Checking for presence and/or absence when adding and/or retrieving tokens

Proposition 11. $\mathcal{L}_B$(get,tell) $<$ $\mathcal{L}_B$(nask,get,tell)

Proof. **(i)** The proof methods of both parts are the same as in [BJ98]. On the one hand $\mathcal{L}_B$(get,tell) $\leqslant$ $\mathcal{L}_B$(nask,get,tell) results from proposition 1. **(ii)** On the other hand, $\mathcal{L}_B$(nask,get,tell) $\not\leqslant$ $\mathcal{L}_B$(get,tell) is established by contradiction, similarly to the reasoning conducted in the second part of proposition 4. Consider agent $tell(t)$; $nask(t)$ with $\mathcal{O}(tell(t)\,;\,nask(t)) = \{(\{t\}, \delta^-)\}$. It is possible to prove that $\mathcal{C}(tell(t)\,;\,nask(t))$ starting in the empty store is successful, which contradicts property P3 of termination invariance. □

Proposition 12. $\mathcal{L}_B$(nask,get,tell) = $\mathcal{L}_B$(ask,nask,get,tell)

Proof. **(i)** On the one hand, $\mathcal{L}_B$(nask,get,tell) $\leqslant$ $\mathcal{L}_B$(ask,nask,get,tell) is established by proposition 1 on language inclusion. **(ii)** On the other hand, to establish $\mathcal{L}_B$(ask,nask,get,tell) $\leqslant$ $\mathcal{L}_B$(nask,get,tell) we shall provide a coder such that the coding of the primitives $ask(t)$ and $nask(t)$ manipulate different tokens. The proof method is the same as the one presented in [BJ98]. As the set of tokens is denumerable, it is possible to associate each of them, say t, to a pair (t_1, t_2) (for instance it suffices to associate the token associated with the integer n to the

tokens associated with the integers $2n$ and $2n+1$. Given such a coding of tokens, we define the compositional coder $\mathcal{C}$ as follows:

$$\begin{array}{lcl} \mathcal{C}(ask(t)) & = & get(t_2)\,;\,tell(t_2) \\ \mathcal{C}(nask(t)) & = & nask(t_1) \\ \mathcal{C}(get(t)) & = & get(t_2)\,;\,get(t_1) \\ \mathcal{C}(tell(t)) & = & tell(t_1)\,;\,tell(t_2) \end{array}$$

The decoder $\mathcal{D}$ is defined as follows: $\mathcal{D}_{el}((\sigma,\delta)) = (\overline{\sigma},\delta)$, where $\overline{\sigma}$ is composed of the tokens t for which t_1 and t_2 are in σ, the multiplicity of t being that of pairs (t_1,t_2) in σ. With those definitions of the coder $\mathcal{C}$ and the decoder $\mathcal{D}$, properties P1 of element-wise and P2 of compositionality are guaranteed. It remains to establish property P3 of termination invariance and that $\mathcal{O}(A) = \mathcal{D}(\mathcal{O}(\mathcal{C}(A)))$ for any agent A of $\mathcal{L}_B$(ask,nask,get,tell). The proof consists of establishing that for any agent A and stores σ and τ:

1. $\langle A|\overline{\sigma}\rangle \rightarrow^* \langle E|\overline{\tau}\rangle$ iff $\langle \mathcal{C}(A)|\sigma\rangle \rightarrow^* \langle E|\tau\rangle$
2. there is some agent B such that $\langle A|\overline{\sigma}\rangle \rightarrow^* \langle B|\overline{\tau}\rangle \nrightarrow$ with $B \neq E$
   iff there is some agent $B' \in \mathcal{L}_B$(nask,get,tell) such that $\langle \mathcal{C}(A)|\sigma\rangle \rightarrow^* \langle B'|\tau\rangle \nrightarrow$

This is proved by induction on the structure of the agent. □

Proposition 13. $\mathcal{L}_B$(ask,nask,tell) $<$ $\mathcal{L}_B$(ask,nask,get,tell)

Proof. **(i)** $\mathcal{L}_B$(ask,nask,tell) $\leqslant$ $\mathcal{L}_B$(ask,nask,get,tell) is immediate by proposition 1. **(ii)** $\mathcal{L}_B$(ask,nask,get,tell) $\not\leqslant$ $\mathcal{L}_B$(ask,nask,tell) is established by contradiction. Suppose that $\mathcal{L}_B$(ask,nask,get,tell) $\leqslant$ $\mathcal{L}_B$(ask,nask,tell). Then, since $\mathcal{L}_B$(get,tell) $\leqslant$ $\mathcal{L}_B$(nask,get,tell) and since $\mathcal{L}_B$(nask,get,tell) $\leqslant$ $\mathcal{L}_B$(ask,nask,get,tell), by proposition 12, we would have that $\mathcal{L}_B$(get,tell) $\leqslant$ $\mathcal{L}_B$(ask,nask,tell) by pattern 2 on embedding by transitivity, which contradicts proposition 10. □

A.2 BachT in comparison with MRT

Proposition 14. For any agent A of BachT or MRT there is an agent in normal form N such that $\mathcal{O}$(N) = $\mathcal{O}$(A).

Proof. The proof has been established in [Lin07], to which we refer the reader. □

A.2.1 Sublanguages

Proposition 15. $\mathcal{L}_B(\,\chi) \leqslant \mathcal{L}_{MR}(\,\chi)$, for any subset of χ of primitives.

Proof. Immediate by defining the coder as follows:

$$\begin{array}{rclcrcl} \mathcal{C}(tell(t)) & = & (\{\},\{+t\}) & & \mathcal{C}(get(t(m))) & = & (\{+t\},\{-t\}) \\ \mathcal{C}(ask(t(m))) & = & (\{+t\},\{\}) & & \mathcal{C}(nask(t(m))) & = & (\{-t\},\{\}) \end{array}$$

and using the identity as decoder. □

A.2.2 Putting tokens on the store

Proposition 16. $\mathcal{L}_B$(tell) and $\mathcal{L}_{MR}$(tell) are equivalent.

Proof. We have $\mathcal{L}_B(\text{tell}) \leqslant \mathcal{L}_{MR}(\text{tell})$ by proposition 15. Furthermore, $\mathcal{L}_{MR}(\text{tell}) \leqslant \mathcal{L}_B(\text{tell})$ is established by coding any tell primitive of $\mathcal{L}_{MR}$(tell) by a sequence of tell primitives of $\mathcal{L}_B$(tell):

$$\mathcal{C}(\{\},\{+t_1,\ldots,+t_n\}) = tell(t_1)\ ;\ \ldots\ ;\ tell(t_n),$$

some of the t_i's being possibly identical. □

A.2.3 Checking for presence and/or absence when adding tokens

Proposition 17. $\mathcal{L}_B$(ask,tell) $< \mathcal{L}_{MR}$(ask,tell)

Proof. Both parts of the proof are established as in [BJ99, BJ03b]. **(i)** On the one hand, $\mathcal{L}_B(\text{ask,tell}) \leqslant \mathcal{L}_{MR}(\text{ask,tell})$, by proposition 15. **(ii)** On the other hand, $\mathcal{L}_{MR}(\text{ask,tell}) \not\leqslant \mathcal{L}_B(\text{ask,tell})$ may be established as in [BJ99, BJ03b], by exploiting the inability of $\mathcal{L}_B$(ask,tell) to atomically test the presence of two distinct tokens a and b. Applying pattern 4 of presence, one considers $AB = (\{+a,+b\},\{\})$ and assumes that $\mathcal{C}(AB)$ is in normal form (see definition 9) and thus is written as

$$tell(t_1); A_1 + \cdots + tell(t_p); A_p + ask(u_1); B_1 + \cdots + ask(u_q); B_q$$

In this expression, we will establish that there is no alternative guarded by a $tell(t_i)$ operation and no alternative guarded by an $ask(u_j)$ operation either, which is impossible since $\mathcal{C}(AB)$ must contain at least one primitive.

Let us first establish by contradiction that there is no alternative guarded by a $tell(t_i)$ operation. Indeed, if there is an alternative guarded, say by $tell(t_i)$, then

$$D = \langle \mathcal{C}(AB) | \varnothing \rangle \rightarrow \langle A_i | t_i \rangle$$

is a valid computation prefix of $\mathcal{C}(AB)$. It should deadlock afterwards since $\mathcal{O}(AB) = (\varnothing, \delta^-)$. However D is also a valid computation prefix of $\mathcal{C}(AB + (\{\},\{+a\}))$. Hence, $\mathcal{C}(AB + (\{\},\{+a\}))$ admits a failing computation which contradicts the fact that $\mathcal{O}(AB + (\{\},\{+a\})) = (\{a\}, \delta^+)$.

Secondly, we establish that there is also no alternative guarded by an $ask(u_j)$ operation. To that end, let us first consider two auxiliary computations. As $\mathcal{O}((\{\},\{+a\})) = (\{a\},\delta^+)$, any computation of $\mathcal{C}((\{\},\{+a\})$ starting in the empty store succeeds. Let

$$\langle \mathcal{C}((\{\},\{+a\}))|\varnothing\rangle \rightarrow \cdots \rightarrow \langle E|\{a_1,\ldots,a_m\}\rangle$$

be such a computation. Similarly, let

$$\langle \mathcal{C}((\{\},\{+b\}))|\varnothing\rangle \rightarrow \cdots \rightarrow \langle E|\{b_1,\ldots,b_n\}\rangle$$

be one computation of $\mathcal{C}((\{\},\{+b\}))$. The proof of the claim proceeds by first establishing that none of the u_i's belong to $\{a_1,\ldots,a_m\} \cup \{b_1,\ldots,b_n\}$.

First let us prove that none of the u_j's belong to $\{a_1,\ldots,a_m\}$. By contradiction, assume that $u_i = a_k$ for some k. Then

$$D' = \langle \mathcal{C}((\{\},\{+a\});AB)|\varnothing\rangle \rightarrow \cdots \rightarrow \langle AB|\{a_1,\ldots,a_m\}\rangle \rightarrow \langle B_j|\{a_1,\ldots,a_m\}\rangle$$

is a valid computation prefix of $\mathcal{C}((\{\},\{+a\});AB)$, which can only be continued by failing suffixes. However D' induces the following computation prefix D'' for $(\{\},\{+a\});(AB + (\{+a\},\{\}))$ which as just seen admits only successful computations:

$$\begin{aligned} D'' = \langle \mathcal{C}((\{\},\{+a\});(AB + (\{+a\},\{\})))|\varnothing\rangle \rightarrow \ldots \\ \rightarrow \langle AB + (\{+a\},\{\})|\{a_1,\ldots,a_m\}\rangle \rightarrow \langle B_j|\{a_1,\ldots,a_m\}\rangle \end{aligned}$$

The proof proceeds similarly in the case some $u_j \in \{b_1,\ldots,b_n\}$ for some $j \in 1,\ldots,q$ by then considering $(\{\},\{+b\});AB$ and $(\{\},\{+b\});(AB + (\{+b\},\{\}))$.

Finally, the fact that the $u_j's$ do not belong to $\{a_1,\ldots,a_m\} \cup \{b_1,\ldots,b_n\}$ induces a contradiction. Indeed, if this is the case, then

$$\begin{aligned} \langle \mathcal{C}((\{\},\{+a\});(\{\},\{+b\});AB)|\varnothing\rangle \rightarrow \ldots \\ \rightarrow \langle (\{\},\{+b\});AB)|\{a_1,\ldots,a_m\}\rangle \rightarrow \\ \cdots \rightarrow \langle AB|\{a_1,\ldots,a_m,b_1,\ldots,b_n\}\rangle \nrightarrow \end{aligned}$$

is a valid failing computation prefix of $\mathcal{C}((\{\},\{+a\});(\{\},\{+b\});AB)$ whereas $(\{\},\{+a\});(\{\},\{+b\});AB$ has only one successful computation. □

Lemma 1. Let $f : Stoken \rightarrow \mathcal{P}_f(Stoken)$ be a function associating each token with a finite set of tokens. Assume that $f(a) \cap f(b) \neq \varnothing$, for any pair of distinct tokens a and b. Then there is a denumerable sequence of distinct tokens x_i's and an integer N such that

$$\bigcap_{i=1}^{N} f(x_i) \neq \varnothing$$

and

$$\bigcap_{i=1}^{N} f(x_i) = \bigcap_{i=1}^{N} f(x_i) \cap f(x_j)$$

for any $j > N$.

Proof. The proof of this technical lemma has been established in [BJ03b], to which we refer the reader. □

Proposition 18. $\mathcal{L}_B$(nask,tell) $< \mathcal{L}_{MR}$(nask,tell).

Proof. **(i)** On the one hand, $\mathcal{L}_B$(nask,tell) $\leqslant \mathcal{L}_{MR}$(nask,tell) holds by proposition 15. **(ii)** On the other hand, $\mathcal{L}_{MR}$(nask,tell) $\not\leqslant \mathcal{L}_B$(nask,tell) is established as in [BJ99, BJ03b], by assuming the existence of a coder $\mathcal{C}$, and by establishing that it contains in fact no primitive, while it has to contain at least one. The proof is similar to part (ii) of proposition 17, but this time by exploiting the inability of $\mathcal{L}_B$(nask,tell) to atomically test the absence of two distinct tokens a and b, following the schema of pattern 5 of absence.

To do so the construction of the tokens $\{a_1, \ldots, a_m\}$ and $\{b_1, \ldots, b_n\}$ associated with the coding of a and b is generalized by the definition of a function $f : Stoken \to \mathcal{P}_f(Stoken)$, associating with each token a finite set of tokens.

For any token t, as $\mathcal{O}((\{\}, \{+t\})) = \{(\{t\}, \delta^+)\}$, any computation of $\mathcal{C}((\{\}, \{+t\}))$ starting in the empty store succeeds. Let $\langle((\{\}, \{+t\}))|\varnothing\rangle \to \cdots \to \langle E|\{t_1, \ldots, t_{m_t}\}\rangle$ be such a computation and let S_t denote the resulting store $\{t_1, \cdots, t_{m_t}\}$.

Then the proof of the claim proceeds by examining two cases: (I) either there exist two (distinct) tokens a and b such that $S_a \cap S_b = \varnothing$, (II) or $S_a \cap S_b \neq \varnothing$ for any pair of (distinct) tokens a and b.

CASE I: Let us first suppose that there are two tokens a and b such that $S_a \cap S_b = \varnothing$. One considers $AB = (\{-a, -b\}, \{\})$ and $\mathcal{C}(AB)$ in its normal form:

$$tell(v_1)\ ;\ A_1 + \cdots + tell(v_p)\ ;\ A_p + nask(u_1)\ ;\ B_1 + \cdots + nask(u_q)\ ;\ B_q$$

The proof then proceeds by establishing that there are no alternatives guarded by $tell(v_i)$ nor by $nask(u_j)$. The absence of alternative guarded by a $tell(v_i)$ primitive is established as in part (ii) of proposition 17: if this was not the case, then AB would point out a deadlocking computation for $(\{\}, \{+a\}); (AB + (\{\}, \{+a\}))$ which only admits successful computations. To prove the absence of alternatives guarded by a $nask(u_j)$ primitive, one establishes that the u_j's should belong to S_a and to S_b, which is impossible since $S_a \cap S_b = \varnothing$. By contradiction, assume that $u_j \notin S_a$ for some j (the case where $u_j \notin S_b$ is treated similarly). Then

$$\langle \mathcal{C}((\{\}, \{+a\})\ ;\ AB) \mid \varnothing\rangle \longrightarrow \cdots \longrightarrow \langle \mathcal{C}(AB) \mid S_a\rangle \longrightarrow \langle B_j \mid S_a\rangle$$

is a valid computation prefix of $\mathcal{C}((\{\},\{+a\})\,;AB)$ which can only be continued by failing suffixes. However, this prefix induces the following computation prefix D' for $\mathcal{C}((\{\},\{+a\})\,;(AB+(\{\},\{+a\}))$ which should only admit successful computations:

$$\langle \mathcal{C}((\{\},\{+a\})\,;(AB+(\{\},\{+b\}))) \mid \varnothing\rangle \longrightarrow \cdots$$

$$\longrightarrow \langle AB+(\{\},\{+b\}) \mid S_a\rangle \longrightarrow \langle B_j \mid S_a\rangle$$

CASE II: Let us now suppose that $S_a \cap S_b \neq \varnothing$ for any pair of tokens a and b. As proved by Lemma 1, it is possible to construct an infinite sequence of distinct tokens x_i's and to identify an integer n such that

$$\bigcap_{i=1}^{n} S_{x_i} \neq \varnothing$$

and

$$\bigcap_{i=1}^{n} S_{x_i} = \bigcap_{i=1}^{n} S_{x_i} \cap S_{x_j}$$

for any $j > n$. Let us consider $NT = (\{-x_1,\cdots,-x_n\},\{\})$ and $\mathcal{C}(NT)$ in its normal form

$$tell(v_1)\,;A_1+\cdots+tell(v_p)\,;A_p+nask(u_1)\,;B_1+\cdots+nask(u_q)\,;B_q$$

By using a reasoning similar to the one employed for case I, one may prove that there are no alternatives guarded by a $tell(v_i)$ primitive and that $\{u_1,\cdots,u_q\} \subseteq S_{x_1}\cap\cdots\cap S_{x_n}$. Therefore $\mathcal{C}((\{\},\{+x_{n+1}\})\,;NT)$ has a failing computation since $S_{x_1}\cap\cdots\cap S_{x_n}\cap S_{x_{n+1}} = S_{x_1}\cap\cdots\cap S_{x_n}$ and thus $\{u_1,\cdots,u_q\} \subseteq S_{x_1}\cap\cdots\cap S_{x_n} \subseteq S_{x_{n+1}}$. However, this contradicts the fact that $(\{\},\{+x_{n+1}\})\,;NT$ has only one successful computation.

In conclusion, $\mathcal{C}(AB)$ reduces to an empty statement, which is not possible since it should contain at least one primitive. □

Proposition 19. $\mathcal{L}_{MR}$(ask,tell) $\wr$ $\mathcal{L}_B$(nask,tell)

Proof. **(i)** On the one hand, we have that $\mathcal{L}_{MR}$(ask,tell) $\nleqslant$ $\mathcal{L}_B$(nask,tell), otherwise, by pattern 3 of non embedding by transitivity, we have $\mathcal{L}_B$(ask,tell) $\leqslant$ $\mathcal{L}_B$(nask,tell) which has been proved impossible in proposition 4. **(ii)** On the other hand, $\mathcal{L}_B$(nask,tell) $\nleqslant$ $\mathcal{L}_{MR}$(ask,tell) is established as in [BJ98] by contradiction, by considering $tell(t)\,;nask(t)$ with $\mathcal{O}(tell(t)\,;nask(t)) = \{(\{t\},\delta^-)\}$ and by employing the reasoning developped in part (ii) of the proof of proposition 4 of section A.1. □

Proposition 20. $\mathcal{L}_{MR}$(nask,tell) $\wr$ $\mathcal{L}_B$(ask,tell)

Proof. **(i)** On the one hand, we have that $\mathcal{L}_{MR}$(nask,tell) $\nleqslant$ $\mathcal{L}_B$(ask,tell). Otherwise, by pattern 3 of non embedding by transitivity, we have $\mathcal{L}_B$(nask,tell) $\leqslant$ $\mathcal{L}_B$(ask,tell) which has been

proved impossible in proposition 4. **(ii)** On the other hand, $\mathcal{L}_B$(ask,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(nask,tell) is established as in [BJ98] by contradiction, by considering $tell(t)$; $ask(t)$, with $\mathcal{O}(tell(t)$; $ask(t)) = \{(\{t\}, \delta^+)\}$, and using the reasoning developped in part (i) of the proof of proposition 4 of section A.1. □

Proposition 21. $\mathcal{L}_B$(ask,nask,tell) $\wr$ $\mathcal{L}_{MR}$(nask,tell)

Proof. **(i)** On the one hand, we have that $\mathcal{L}_B$(ask,nask,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(nask,tell), otherwise, by pattern 3 of non embedding by transitivity, $\mathcal{L}_B$(ask,tell) $\leqslant$ $\mathcal{L}_{MR}$(nask,tell), which has been proved impossible in proposition 20. **(ii)** On the other hand, $\mathcal{L}_{MR}$(nask,tell) $\not\leqslant$ $\mathcal{L}_B$(ask,nask,tell) is established as in [BJ99, BJ03b]. The proof is an extension of part (ii) of the proof of proposition 18, that establishes that $\mathcal{L}_{MR}$(nask,tell) $\not\leqslant$ $\mathcal{L}_B$(nask,tell), with normal forms extended with *ask* primitives. Using the notations of this proof, and following the proof technique 2, we thus examine two cases and conclude for each one by a contradiction: (I) either there exist two tokens a and b such that $S_a \cap S_b = \varnothing$, (II) or $S_a \cap S_b \neq \varnothing$ for any pair of tokens a and b. This proof follows the schema of pattern 5 of absence.

CASE I: Let us suppose that there are two tokens a and b such $S_a \cap S_b = \varnothing$. Consider $AB = (\{-a, -b\}, \{\})$ and $\mathcal{C}(AB)$ in its normal form:

$$\begin{array}{rl} & tell(t_1)\ ;\ A_1 + \cdots + tell(t_p)\ ;\ A_p \\ + & ask(u_1)\ ;\ B_1 + \cdots + ask(u_q)\ ;\ B_q \\ + & nask(v_1)\ ;\ C_1 + \cdots + nask(v_r)\ ;\ C_r \end{array}$$

The proof then proceeds by establishing that there are no alternatives guarded by *tell* and *nask* primitives. In that case, $\mathcal{C}(AB)$ reduces to

$$ask(u_1)\ ;\ B_1 + \cdots + ask(u_q)\ ;\ B_q$$

which thus fails on the empty store whereas $\mathcal{O}(AB) = \{(\varnothing, \delta^+)\}$, providing the contradiction. The absence of alternatives guarded by a $tell(t_i)$ primitive is established as in part (ii) of the proof of proposition 17: if this was not the case then AB would point out a deadlocking computation for $(\{\}, \{+a\}); (AB + (\{\}, \{+a\}))$ which only admits successful computations. The absence of alternatives guarded by a $nask(v_i)$ primitive is established as in part (ii) of proposition 18, namely by establishing that the v_i's should belong to S_a and to S_b, which is impossible since $S_a \cap S_b = \varnothing$.

CASE II: In the case where $S_a \cap S_b \neq \varnothing$ for any pair of tokens a and b, as proved by lemma 1, it is possible to construct an infinite sequence of distinct tokens x_i's and to identify an integer N such that

$$\bigcap_{i=1}^{N} S_{x_i} \neq \varnothing$$

and

$$\bigcap_{i=1}^{N} S_{x_i} = \bigcap_{i=1}^{N} S_{x_i} \cap S_{x_j}$$

for any $j > N$. Consider now $NT = (\{-x_1, \cdots, -x_n\}, \{\})$ and $\mathcal{C}(NT)$ in its normal form

$$\begin{array}{rl} & tell(t_1)\ ;\ A_1 + \cdots + tell(t_p)\ ;\ A_p \\ + & ask(u_1)\ ;\ B_1 + \cdots + ask(u_q)\ ;\ B_q \\ + & nask(v_1)\ ;\ C_1 + \cdots + nask(v_r)\ ;\ C_r \end{array}$$

By reasoning similarly to case I, one may prove that there are no alternatives guarded by a $tell(t_i)$ primitive. As regards the ask and nask primitives, the proof proceeds by contradiction and establishes successively that

$$\{u_1, \cdots, u_q\} \cap (S_{x_1} \cup \cdots \cup S_{x_n}) = \varnothing,$$

and that

$$\{v_1, \cdots, v_r\} \subseteq (S_{x_1} \cap \cdots \cap S_{x_n})$$

and derive a contradiction therefrom.

Let us first establish that $\{u_1, \cdots, u_q\} \cap (S_{x_1} \cup \cdots \cup S_{x_n}) = \varnothing$. By contradiction, assume $u_j \in S_{x_i}$, for some i, j. Consequently,

$$F = \langle (\mathcal{C}((\{\}, \{+x_i\}))) \ ;\ \mathcal{C}(NT) \mid \varnothing \rangle \longrightarrow \cdots$$

$$\longrightarrow \langle \mathcal{C}(NT) \mid S_{x_i} \rangle \longrightarrow \langle B_j \mid S_{x_i} \rangle$$

is a valid computation prefix for $(\mathcal{C}((\{\}, \{+x_i\}))) \ ;\ \mathcal{C}(NT))$ which can only be continued by failing suffixes. However F is also a computation prefix for $\mathcal{C}((\{\}, \{+x_i\}) \ ;\ (NT + (\{+x_i\}, \{\})))$ which thus induces a failing computation for it whereas $(\{\}, \{+x_i\}) \ ;\ (NT + (\{+x_i\}, \{\})))$ has only one successful computation.

Let us now establish that $\{v_1, \cdots, v_r\} \subseteq (S_{x_1} \cap \cdots \cap S_{x_n})$. Assume $v_k \notin S_{x_i}$, for some k, i. Then

$$F' = \langle \mathcal{C}((\{\}, \{+x_i\})) \ ;\ \mathcal{C}(NT) \mid \varnothing \rangle \longrightarrow \cdots \longrightarrow \langle \mathcal{C}(NT) \mid S_{x_i} \rangle \longrightarrow \langle C_k \mid S_{x_i} \rangle$$

is a valid computation prefix for $\mathcal{C}((\{\}, \{+x_i\}) \ ;\ NT)$ which can only be continued by failing suffixes. However, it is also a computation prefix for $(\{\}, \{+x_i\}) \ ;\ (NT + (\{+x_i\}, \{\})))$, for which it thus induces a failing computation whereas $(\{\}, \{+x_i\}) \ ;\ (NT + (\{+x_i\}, \{\})))$ has only one successful computation.

In order to establish the final contradiction, let us consider $\mathcal{C}((\{\}, \{+x_{n+1}\}) \ ;\ NT)$. A possible computation prefix is as follows:

$$\langle \mathcal{C}((\{\}, \{+x_{n+1}\})) \ ;\ \mathcal{C}(NT) \mid \varnothing \rangle \longrightarrow \cdots \longrightarrow \langle \mathcal{C}(NT) \mid S_{x_{n+1}} \rangle.$$

Since $(\{\},\{+x_{n+1}\})$; NT has a successful computation and since, $\{v_1,\cdots,v_r\} \subseteq S_{x_1}\cap\cdots\cap S_{x_n} \subseteq S_{x_{n+1}}$ by the choice of the x_i's and the above inclusion, we can exclude the execution of a *nask* primitive and therefore force the execution of an *ask* primitive and thus the existence of a j such that $u_j \in S_{x_{n+1}}$. Now consider

$$(\{\},\{+x_{n+1}\}) ; (\{\},\{+x_1\}) ; NT$$

which has failing computations only. At the coded level, since $\mathcal{L}_B$(ask,nask,tell) does not contain any destructive primitive, the computation of $\mathcal{C}((\{\},\{+x_1\}))$ can only enrich the store resulting from the computation of $\mathcal{C}((\{\},\{+x_{n+1}\}))$, say by some set of tokens σ. Consequently, the following derivation sequence G is a valid computation prefix for $\mathcal{C}((\{\},\{+x_{n+1}\}) ; (\{\},\{+x_1\}) ; NT)$, which should be continued by failing suffixes only:

$$\begin{aligned} G \;=\; & \langle \mathcal{C}((\{\},\{+x_{n+1}\})) ; \mathcal{C}((\{\},\{+x_1\})) ; \mathcal{C}(NT) \mid \varnothing\rangle \longrightarrow \cdots \\ & \longrightarrow \langle \mathcal{C}((\{\},\{+x_1\})) ; \mathcal{C}(NT) \mid S_{x_{n+1}}\rangle \longrightarrow \cdots \\ & \longrightarrow \langle \mathcal{C}(NT) \mid S_{x_{n+1}} \cup \sigma\rangle \longrightarrow \langle B_j \mid S_{x_{n+1}} \cup \sigma\rangle \end{aligned}$$

However, G is also a computation prefix G' for

$$\mathcal{C}((\{\},\{+x_{n+1}\}) ; (\{\},\{+x_1\}) ; (NT + (\{+x_{n+1}\},\{\})))$$

which thus induces a failing computation for

$$(\{\},\{+x_{n+1}\}) ; (\{\},\{+x_1\}) ; (NT + (\{+x_{n+1}\},\{\}))$$

which is impossible since it admits only a successful computation. □

A.2.4 Retrieving tokens from the store in the BachT language

Proposition 22. $\mathcal{L}_B$(get,tell) $\wr$ $\mathcal{L}_{MR}$(ask,tell)

Proof. **(i)** On the one hand, the reasoning to prove that $\mathcal{L}_B$(get,tell) $\nleqslant$ $\mathcal{L}_{MR}$(ask,tell) is the same as the one used in part (ii) of the proof of proposition 7, as established in [BJ98]. It works by contradiction and establishes that $tell(t)$; $get(t)$ can produce a successful computation for $\mathcal{C}(tell(t))$; $\mathcal{C}(get(t))$; $\mathcal{C}(get(t))$, leading to an obvious contradiction. **(ii)** On the other hand, $\mathcal{L}_{MR}$(ask,tell) $\nleqslant$ $\mathcal{L}_B$(get,tell) is established as in [BJ99, BJ03b]. Intuitively, $\mathcal{L}_B$(get,tell) is unable to atomically test the presence of a and b. The proof applies pattern 4 of presence. Let us thus consider $AB = (\{+a,+b\},\{\})$ and assume that $\mathcal{C}(AB)$ is in normal form (see definitin 9)) and thus is written as

$$tell(t_1); A_1 + \cdots + tell(t_p); A_p + get(u_1); B_1 + \cdots + get(u_q); B_q$$

The proof proceeds as explained in proof technique 1 by establishing (I) that there is no alternative guarded by a $tell(t_i)$ operation, and (II) that there is no alternative guarded by a $get(u_j)$

operation, in which case, $\mathcal{C}(AB)$ is equivalent to an empty statement, which is impossible since it is composed of at least one primitive.

STEP I: Let us first establish that there is no existence of an alternative guarded by a $tell(t_i)$ operation. Otherwise it would point out a failing computation for $\mathcal{C}(AB + (\{\}, \{+a\}))$, contradicting the fact that $\mathcal{O}(AB + (\{\}, \{+a\})) = (\{a\}, \delta^+)$.

STEP II: Let us now establish that there is no alternative guarded by a $get(u_j)$ operation. To that end, let us first consider two auxiliary computations. As $\mathcal{O}((\{\}, \{+a\})) = \{(\{a\}, \delta^+)\}$, any computation of $\mathcal{C}((\{\}, \{+a\}))$ starting in the empty store succeeds. Let

$$\langle (\{\}, \{+a\}) | \varnothing \rangle \rightarrow \cdots \rightarrow \langle E | \{a_1, \ldots, a_m\} \rangle$$

be such a computation. Similarly, let

$$\langle (\{\}, \{+b\}) | \varnothing \rangle \rightarrow \cdots \rightarrow \langle E | \{b_1, \ldots, b_n\} \rangle$$

be one computation of $\mathcal{C}((\{\}, \{+b\}))$.

The proof of the claim proceeds in two steps: none of the u_i's belong to $\{a_1, \ldots, a_m\} \cup \{b_1, \ldots, b_n\}$ but, in that case, a contradiction occurs from the analysis of $\mathcal{C}((\{\}, \{+a\}); (\{\}, \{+b\}); AB)$. As a result, none of the u_i's exist, namely there is no alternative guarded by a $get(u_j)$ operation. □

Proposition 23. $\mathcal{L}_B$(nask,get,tell) $\wr$ $\mathcal{L}_{MR}$(ask,tell)

Proof. **(i)** On the one hand, $\mathcal{L}_B$(nask,get,tell) $\nleqslant$ $\mathcal{L}_{MR}$(ask,tell). Otherwise, by pattern 3 of non embedding by transitivity, $\mathcal{L}_B$(nask,tell) $\leqslant$ $\mathcal{L}_{MR}$(ask,tell) which has been proved impossible in part (ii) of proposition 19. **(ii)** On the other hand, $\mathcal{L}_{MR}$(ask,tell) $\nleqslant$ $\mathcal{L}_B$(nask,get,tell) is established as in [BJ99, BJ03b]. The intuition behind the proof is again that $\mathcal{L}_B$(nask,get,tell) is not able to test atomically the presence of two distinct tokens a and b. Following pattern 4 of presence, we then proceed by contradiction using these two tokens a and b. However, the destructive character of *get* primitives coupled to the test for absence of *nask* slightly complicate our task of producing a contradiction. To that end, we shall "saturate" their effect by taking as many instances of codings in parallel and thereby by extending the sets S_b introduced in part (ii) of the proof of proposition 18.

Let us thus proceed by contradiction by assuming the existence of a coder $\mathcal{C}$. Take two distinct tokens a and b. Let n be the number of occurences of *nask* and *get* primitives of $\mathcal{C}((\{\}, \{+a\}))$. As $\mathcal{C}((\{\}, \{+a\}))$ has only successful computations, let, as in the part (ii) of the proof of proposition 18, S_a be the store resulting from one of them. As $(||_{k=1}^{n+2}(\{\}, \{+b\}))\ ;\ (\{\}, \{+a\}))$ succeeds as well, let S'_b denote the store resulting from one successful computation of its coding. Consider finally $ABs = (\{+a, +b, \cdots, +b\}, \{\})$ requesting one

a with $n+3$ copies of b and $\mathcal{C}(ABs)$ in its normal form:

$$\begin{array}{l} tell(t_1)\ ;\ A_1 + \cdots + tell(t_p)\ ;\ A_p \\ + get(u_1)\ ;\ B_1 + \cdots + get(u_q)\ ;\ B_q \\ + nask(v_1)\ ;\ C_1 + \cdots + nask(v_r)\ ;\ C_r \end{array}$$

Following proof technique 1, we shall establish (I) that there are no alternatives guarded by *tell* and *nask* primitives, and (II) that $\{u_1, \cdots, u_q\} \cap (S_a \cup S'_b) = \varnothing$. Assuming these two points proved, a contradiction can be produced as follows. In view of the saturation provided by the $n+2$ copies of $\mathcal{C}((\{\},\{+b\}))$, adding one more only adds tokens present in $S_a \cup S'_b$. As a result, $\mathcal{C}(||_{k=1}^{n+3}(\{\},\{+b\}))\ ;\ (\{\},\{+a\})\ ;\ ABs)$ fails whereas $||_{k=1}^{n+3}(\{\},\{+b\}))\ ;\ (\{\},\{+a\})\ ;\ ABs$ has only one successful computation. Hence the contradiction.

STEP I: Let us first establish that there are no alternative guarded by a $tell(t_i)$ primitive. The proof proceeds by contradiction as in part (ii) of the proof of proposition 17, by pointing out a failing computation for $\mathcal{C}(AB + (\{\},\{+a\}))$, contradicting the fact that $\mathcal{O}(AB + (\{\},\{+a\})) = (\{a\}, \delta^+)$.

In a similar way there are no alternative guarded by a *nask* primitive. Indeed assuming the existence of a $nask(v_i)\ ;\ C_i$ alternative again points out a failing computation for $\mathcal{C}(AB + (\{\},\{+a\}))$, contradicting the fact that $\mathcal{O}(AB + (\{\},\{+a\})) = (\{a\}, \delta^+)$.

STEP II: Let us now establish that $\{u_1, \cdots, u_q\} \cap (S_a \cup S'_b) = \varnothing$. This is proved in two steps by establishing (1) that $\{u_1, \cdots, u_q\} \cap S_a = \varnothing$, and (2) that $\{u_1, \cdots, u_q\} \cap S'_b = \varnothing$.

First we have that $\{u_1, \cdots, u_q\} \cap S_a = \varnothing$. By contradiction, assume that $u_i \in S_a$ for some i. Let us observe that each step of the considered computation of $\mathcal{C}((\{\},\{+a\}))$ can be repeated in turn, in as many parallel occurences of it as needed, so that

$$\begin{array}{rl} P = \langle \mathcal{C}((||_{k=1}^{q}(\{\},\{+a\}))\ ;\ ABs) | \varnothing \rangle & \\ \rightarrow \cdots \rightarrow & \langle ABs |\ \cup_{k=1}^{q} S_a \rangle \\ \rightarrow & \langle B_i | (\cup_{k=1}^{q} S_a) \backslash \overline{u_i} \rangle \end{array}$$

is a valid computation prefix of $\mathcal{C}((||_{k=1}^{q}(\{\},\{+a\}))\ ;\ ABs)$, which can only be continued by failing suffixes. However P induces the following computation prefix P' for $\mathcal{C}((||_{k=1}^{q}(\{\},\{+a\}))\ ;\ (ABs + (\{\},\{+a\})))$ which admits only successful computations:

$$\begin{array}{rl} P' = \langle \mathcal{C}((||_{k=1}^{q}(\{\},\{+a\}))\ ;\ (ABs + (\{\},\{+a\}))) | \varnothing \rangle & \\ \rightarrow \cdots \rightarrow & \langle \mathcal{C}(ABs + (\{\},\{+a\})) |\ \cup_{k=1}^{q} S_a \rangle \\ \rightarrow & \langle B_i | (\cup_{k=1}^{q} S_a) \backslash \overline{u_i} \rangle \end{array}$$

Hence the contradiction.

The fact that $\{u_1, \cdots, u_q\} \cap S'_b = \varnothing$ is proved similarly, by considering S'_b instead of S_a and $(\{\},\{+b\})$ instead of $(\{\},\{+a\})$. □

Proposition 24. $\mathcal{L}_{MR}$(ask,tell) $\wr$ $\mathcal{L}_{B}$(ask,nask,tell)

Proof. **(i)** On the one hand, one has $\mathcal{L}_{B}$(ask,nask,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(ask,tell). Indeed otherwise by pattern 3, $\mathcal{L}_{B}$(nask,tell) $\leqslant$ $\mathcal{L}_{B}$(ask,nask,tell) $\leqslant$ $\mathcal{L}_{MR}$(ask,tell) which has been proved impossible in part (i) of proposition 19. **(ii)** On the other hand, $\mathcal{L}_{MR}$(ask,tell) $\not\leqslant$ $\mathcal{L}_{B}$(ask,nask,tell) may be established as in [BJ99, BJ03b]. Let us proceed by contradiction by assuming the existence of a coder $\mathcal{C}$. Let us fix two distinct tokens a and b and let n be the number of the *nask* primitives of $\mathcal{C}((\{\},\{+a\}))$.

As $\mathcal{C}((\{\},\{+a\}))$ has only successful computations, let, as in the proof of part (ii) of proposition 18, S_a be the store resulting from one of them. As $(||_{k=1}^{n+2}(\{\},\{+b\}))$; $(\{\},\{+a\}))$ succeeds as well, let S'_b denote the store resulting from one successful computation of its coding. Consider finally $ABs = (\{+a,+b,\cdots,+b\},\{\})$ requesting one a with $n+3$ copies of b and $\mathcal{C}(ABs)$ in its normal form:

$$\begin{array}{rl} & tell(t_1)\ ;\ A_1 + \cdots + tell(t_p)\ ;\ A_p \\ + & ask(u_1)\ ;\ B_1 + \cdots + ask(u_q)\ ;\ B_q \\ + & nask(v_1)\ ;\ C_1 + \cdots + nask(v_r)\ ;\ C_r \end{array}$$

We shall establish, as explained in proof technique 1, (I) that there are no alternatives guarded by $tell(t_i)$ and $nask(v_j)$ primitives, and (II) that $\{u_1,\cdots,u_q\} \cap (S_a \cup S'_b) = \varnothing$. Assuming these facts proved, as repeating once more $\mathcal{C}((\{\},\{+b\}))$ just add copies of tokens already present in $S_a \cup S'_b$, it follows that $\mathcal{C}((((\{\},\{+b\}))^{n+3}\ ;\ (\{\},\{+a\})\ ;\ ABs)$ fails, which is absurd by P_3, since, by construction, $((\{\},\{+b\}))^{n+3}\ ;\ (\{\},\{+a\})\ ;\ ABs$ has one successful computation. Hence the announced contradiction.

STEP I: As for proposition 23, the proof to establish that there are neither alternatives guarded by a *tell* primitive, nor alternatives guarded by a *nask* primitive, proceeds by contradiction. Assuming respectively the existence of a $tell(t_i)\ ;\ A_i$ alternative or a $nask(v_i)\ ;\ C_i$ alternative will in both cases point out a contradiction a failing computation for $\mathcal{C}(AB + (\{\},\{+a\}))$, contradicting the fact that $\mathcal{O}(AB + (\{\},\{+a\})) = (\{a\},\delta^+)$.

STEP II: Let us now prove that $\{u_1,\cdots,u_q\} \cap (S_a \cup S'_b) = \varnothing$. This is established in two steps by demonstrating (1) that $\{u_1,\cdots,u_q\} \cap S_a = \varnothing$, and (2) that $\{u_1,\cdots,u_q\} \cap S'_b = \varnothing$.

First let us prove that $\{u_1,\cdots,u_q\} \cap S_a = \varnothing$. Let us observe that each step of the considered computation of $\mathcal{C}((\{\},\{+a\}))$ can be repeated in turn, in as many parallel occurences of it as needed, so that

$$\begin{array}{rcl} P = \langle \mathcal{C}((||_{k=1}^{q}(\{\},\{+a\}))\ ;\ ABs) | \varnothing \rangle & & \\ \rightarrow \cdots \rightarrow & \langle ABs | \cup_{k=1}^{q} S_a \rangle & \\ \rightarrow & \langle B_i | (\cup_{k=1}^{q} S_a) \backslash \overline{u_i} \rangle & \end{array}$$

is a valid computation prefix of $\mathcal{C}((||_{k=1}^{q}(\{\},\{+a\}))\,;ABs)$, which can only be continued by failing suffixes. However P induces the following computation prefix P' for $\mathcal{C}((||_{k=1}^{q}(\{\},\{+a\}))\,;(ABs+(\{\},\{+a\})))$ which admits only successful computations:

$$\begin{array}{rl} P' = \langle \mathcal{C}((||_{k=1}^{q}(\{\},\{+a\}))\,;(ABs+(\{\},\{+a\})))|\varnothing\rangle & \\ \rightarrow\cdots\rightarrow & \langle \mathcal{C}(ABs+(\{\},\{+a\}))|\cup_{k=1}^{q} S_a\rangle \\ \rightarrow & \langle B_i|(\cup_{k=1}^{q} S_a)\backslash\overline{u_i}\rangle \end{array}$$

Hence the contradiction.

Secondly, the proof that $\{u_1,\cdots,u_q\}\cap S'_b=\varnothing$ is established similarly by considering S'_b instead of S_a and $(\{\},\{+b\})$ instead of $(\{\},\{+a\})$. □

Proposition 25. $\mathcal{L}_B$(ask,nask,tell) $<$ $\mathcal{L}_{MR}$(ask,nask,tell)

Proof. **(i)** On the one hand, the fact that $\mathcal{L}_B$(ask,nask,tell) $\leqslant$ $\mathcal{L}_{MR}$(ask,nask,tell) is immediate by considering the following coder:

$$\begin{array}{rcl} \mathcal{C}(ask(t)) & = & (\{+t\},\{\}) \\ \mathcal{C}(nask(t)) & = & (\{-t\},\{\}) \\ \mathcal{C}(tell(t)) & = & (\{\},\{+t\}) \end{array}$$

(ii) On the other hand, $\mathcal{L}_{MR}$(ask,nask,tell) $\nleqslant$ $\mathcal{L}_B$(ask,nask,tell). Otherwise, by pattern 3 of non embedding by transitivity, from $\mathcal{L}_{MR}$(nask,tell) $\leqslant$ $\mathcal{L}_{MR}$(ask,nask,tell), one would get $\mathcal{L}_{MR}$(nask,tell) $\leqslant$ $\mathcal{L}_B$(ask,nask,tell), which has been proved impossible in proposition 21. □

Lemma 2. Let S be a finite set of tokens. Let f : $Stoken \rightarrow \mathcal{P}_f(Stoken)$ be a function associating to each token a finite set of tokens. Assume that $S\cap f(x)\neq\varnothing$, for any token x. Then there is a denumerable sequence of distinct tokens x_i's and an integer N such that

$$\bigcap_{i=1}^{N}(S\cap S_{x_i})\neq\varnothing$$

and

$$\bigcap_{i=1}^{N}(S\cap S_{x_i})=\bigcap_{i=1}^{N}(S\cap S_{x_i})\cap S_{x_j}$$

for any $j>N$. In particular, $(\bigcap_{i=1}^{N}(S\cap S_{x_i}))\cap(S\cap S_{x_j})\neq\varnothing$, for any $j>N$.

Proof. Admitted result, see [BJ03b]. □

Proposition 26. $\mathcal{L}_B$(nask,get,tell) $\wr$ $\mathcal{L}_{MR}$(nask,tell)

Proof. **(i)** On the one hand, $\mathcal{L}_B$(nask,get,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(nask,tell). Otherwise, by pattern 3 on non embedding by transitivity, $\mathcal{L}_B$(ask,tell) $\leqslant$ $\mathcal{L}_{MR}$(nask,tell) which contradicts part (ii) of the proof of proposition 20. **(ii)** On the other hand, $\mathcal{L}_{MR}$(nask,tell) $\not\leqslant$ $\mathcal{L}_B$(nask,get,tell) is established as in [BJ03b] by contradiction, similarly to the proofs of $\mathcal{L}_{MR}$(nask,tell) $\not\leqslant$ $\mathcal{L}_B$(ask,nask,tell) of proposition 21, which itself extends that of proposition 18.

Given the destructive character of get primitives, we shall enrich them with the saturation technique of the proof of part (ii) of proposition 23 which technically leads to considering the set S'_b instead of the set S_b defined in the proof of the proposition 18. Following proof-technique 2 and using these notations, we thus fix a token a and reason on two cases, both leading to a contradiction: (I) either there exists a token b such that $S_a \cap S'_b = \varnothing$, (II) or, for any token b, one has $S_a \cap S'_b \neq \varnothing$.

CASE I: there is a token b such that $S_a \cap S'_b = \varnothing$. Consider then $AB = (\{-a, -b\}, \{\})$ and $\mathcal{C}(AB)$ in its normal form:

$$\begin{array}{rl} & tell(t_1)\ ;\ A_1 + \cdots + tell(t_p)\ ;\ A_p \\ + & get(u_1)\ ;\ B_1 + \cdots + get(u_q)\ ;\ B_q \\ + & nask(v_1)\ ;\ C_1 + \cdots + nask(v_r)\ ;\ C_r \end{array}$$

As in proposition 17, it is possible to establish that there are no alternatives guarded by a $tell(t_i)$ primitive : if this was the case then, by posing $A = (\{\}, \{+a\})$, the agent AB would point out a deadlock for $A\ ;\ (AB + A)$ which only admits successful computations. As in proposition 23 also, it is possible to establish that the v_i's should belong to S_a and to S'_b, which amounts to stating that there are no alternatives guarded by a $nask(v_j)$ primitive.

Consequently, $\mathcal{C}(AB)$ rewrites as

$$get(u_1)\ ;\ B_1 + \cdots + get(u_q)\ ;\ B_q$$

and thus $\mathcal{O}(\mathcal{C}(AB)) = \{(\varnothing, \delta^-)\}$ which, by P_3, contradicts the fact that $\mathcal{O}(AB) = \{(\varnothing, \delta^+)\}$.

CASE II: for any token b, one has $S_a \cap S'_b \neq \varnothing$. By Lemma 2 (where S_a plays the role of S and f is defined by $f(x) = S'_x$), there exists a denumerable set of distinct tokens x_i, also distinct from a, and an integer m, such that

$\cap_{i=1}^{m}(S_a \cap S'_{x_i}) \neq \varnothing$ and $[\cap_{i=1}^{m}(S_a \cap S'_{x_i})] \cap (S_a \cap S'_{x_j}) \neq \varnothing$, for $j > m$.

Consider $NT = (\{-a, -x_1, \cdots, -x_m\}, \{\})$ and $\mathcal{C}(NT)$ in the following normal form:

$$\begin{array}{l} tell(t_1)\ ;\ A_1 + \cdots + tell(t_p)\ ;\ A_p \\ + get(u_1)\ ;\ B_1 + \cdots + get(u_q)\ ;\ B_q \\ + nask(v_1)\ ;\ C_1 + \cdots + nask(v_r)\ ;\ C_r \end{array}$$

As for case I, it is possible to prove that there are no alternatives guarded by a $tell(t_i)$ primitive. It is also possible to establish that

$$\{v_1, \cdots, v_r\} \subseteq S_a \cap S'_{x_1} \cap \cdots \cap S'_{x_m}$$

Firstly, we have that $v_k \in S_a$, for any k. Otherwise, assume $v_k \notin S_a$, for some k. Then

$$\begin{aligned} F = \ & \langle \mathcal{C}((\{\}, \{+a\})) \,;\, \mathcal{C}(NT) \mid \varnothing \rangle \longrightarrow \cdots \\ & \longrightarrow \langle \mathcal{C}(NT) \mid S_a \rangle \longrightarrow \langle C_k \mid S_a \rangle \end{aligned}$$

is a valid computation prefix for $\mathcal{C}((\{\}, \{+a\}) \,;\, NT)$ which, by property P_3, can only be continued by failing suffixes. However F induces the following computation prefix F' for $\mathcal{C}((\{\}, \{+a\}) \,;\, (NT + (\{\}, \{+a\})))$, and thus a failing computation for it, which by P_3 contradicts the fact that $(\{\}, \{+a\}) \,;\, (NT + (\{\}, \{+a\})))$ has only one successful computation.

Secondly, we have that $v_k \in S'_{x_i}$, for any k and i. By contradiction, assume that $v_k \notin S'_{x_i}$, for some k and i. The proof proceeds similarly by considering $(PP \,;\, NT)$ instead of $(\{\}, \{+a\}) \,;\, NT)$ and $PP \,;\, (NT + (\{\}, \{+x_i\}))$ instead of $(\{\}, \{+a\}) \,;\, (NT + (\{\}, \{+a\}))$ with PP being defined as the parallel composition of $n + 2$ occurrences of $(\{\}, \{+x_i\})$ followed by $(\{\}, \{+a\})$. To that end, note that the computation of $\mathcal{C}(PP)$ leads to the store S'_{x_i} (see the proof of proposition 23).

Consider now $(\{\}, \{+x_{m+1}\}) \,;\, NT$. A possible computation prefix for $\mathcal{C}((\{\}, \{+x_{m+1}\}) \,;\, NT)$ is, by P_2, as follows:

$$\langle \mathcal{C}((\{\}, \{+x_{m+1}\})) \,;\, \mathcal{C}(NT) \mid \varnothing \rangle \longrightarrow^* \langle \mathcal{C}(NT) \mid S_{x_{m+1}} \rangle$$

Since $(\{\}, \{+x_{m+1}\}) \,;\, NT$ has a successful computation, and since $\{v_1, \cdots, v_r\} \subseteq S_a \cap S_{x_1} \cap \cdots \cap S_{x_m} \subseteq S_{x_{m+1}}$ there should exist j such that $u_j \in S_{x_{m+1}}$.

Therefore, as $S_{x_{m+1}} \subseteq S'_{x_{m+1}}$, the following derivation is valid:

$$\begin{aligned} H = \ & \langle \mathcal{C}((\ \|_{k=1}^{n+2}(\{\}, \{+x_{m+1}\}))) \,;\, \mathcal{C}((\{\}, \{+a\})) \,;\, \mathcal{C}(NT) \mid \varnothing \rangle \\ & \longrightarrow^* \langle \mathcal{C}(NT) \mid S'_{x_{m+1}} \rangle \\ & \longrightarrow \langle B_j \mid S'_{x_{m+1}} \backslash \{u_j\} \rangle \end{aligned}$$

Moreover, H should be continued by failing suffixes only since $(\ \|_{k=1}^{n+2}(\{\}, \{+x_{m+1}\})) \,;\, (\{\}, \{+a\}) \,;\, NT$ fails. However, by P_3, this introduces failing computations for $(\ \|_{k=1}^{n+2}(\{\}, \{+x_{m+1}\})) \,;\, (\{\}, \{+a\}) \,;\, (NT + (\{\}, \{+a\}))$ whereas this agent has only one successful computation. □

Proposition 27. $\mathcal{L}_{MR}$(ask,nask,tell) $\wr$ $\mathcal{L}_B$(nask,get,tell)

Proof. **(i)** On the one hand, $\mathcal{L}_{MR}$(ask,nask,tell) $\not\leqslant$ $\mathcal{L}_B$(nask,get,tell). Otherwise, by pattern 3. $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_B$(nask,get,tell) which contradicts proposition 23. **(ii)** On the other hand, $\mathcal{L}_B$(nask,get,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(ask,nask,tell) is established as in [BJ98] and part (ii) of proposition 7,

but by replacing the sequential composition of the `get(t)` primitives by a parallel composition. By contradiction, consider $tell(t)\,;\,get(t)$. $\mathcal{O}((tell(t)\,;\,get(t)) = \{(\varnothing, \delta^+)\}$. Hence any computation of $\mathcal{C}(tell(t))\,;\,\mathcal{C}(get(t))$ is successful. Such a computation is composed of a computation for $\mathcal{C}(tell(t))$ followed by a computation for $\mathcal{C}(get(t))$. As $\mathcal{C}(get(t))$ is composed of ask, nask, tell primitives which do not destroy elements on the store, the latter computation can be repeated step by step which yields successful computation for $\mathcal{C}(tell(t))\,;\,(\mathcal{C}(get(t)) \,||\, \mathcal{C}(get(t)))$. However, $\mathcal{O}(tell(t)\,;\,(get(t) \,||\, get(t)) = \{(\varnothing, \delta^-)\}$. □

Proposition 28. $\mathcal{L}_B$(get,tell) $\wr$ $\mathcal{L}_{MR}$(nask,tell)

Proof. **(i)** On the one hand, $\mathcal{L}_B$(get,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(nask,tell). Otherwise, by pattern 3, as $\mathcal{L}_B$(ask,tell) $<$ $\mathcal{L}_B$(get,tell) (see part (i) of proposition 7), one would have $\mathcal{L}_B$(ask,tell) $\leqslant$ $\mathcal{L}_B$(get,tell) $\leqslant$ $\mathcal{L}_{MR}$(nask,tell) which has been proved impossible in part (ii) of the proof of proposition 20. **(ii)** On the other hand, $\mathcal{L}_{MR}$(nask,tell) $\not\leqslant$ $\mathcal{L}_B$(get,tell). Otherwise, by pattern 3, we would have that $\mathcal{L}_{MR}$(nask,tell) $\leqslant$ $\mathcal{L}_B$(get,tell) $\leqslant$ $\mathcal{L}_B$(nask,get,tell) which has been proved impossible in proposition 26. □

Proposition 29. $\mathcal{L}_B$(get,tell) $\wr$ $\mathcal{L}_{MR}$(ask,nask,tell)

Proof. **(i)** On the one hand, $\mathcal{L}_B$(get,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(ask,nask,tell). The proof proceeds as for establishing that $\mathcal{L}_B$(get,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(ask,tell) in proposition 22, by considering $tell(t)\,;\,get(t)$ and $tell(t)\,;\,(get(t) \,||\, get(t))$, the parallel composition $\mathcal{C}(get(t)) \,||\, \mathcal{C}(get(t))$ repeating in turn each step of $\mathcal{C}(get(t))$. This demonstration proceeds as the one used for part (ii) of the proof of proposition 10, being itself an adpatation of part (ii) of proposition 7. **(ii)** On the other hand, $\mathcal{L}_{MR}$(ask,nask,tell) $\not\leqslant$ $\mathcal{L}_B$(get,tell). Otherwise, by pattern 3, one would have that $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_{MR}$(ask,nask,tell) $\leqslant$ $\mathcal{L}_B$(get,tell) which has been proved impossible in proposition 22. □

A.2.5 Retrieving tokens from the store in MRT

Proposition 30. $\mathcal{L}_B$(get,tell) $<$ $\mathcal{L}_{MR}$(get,tell)

Proof. **(i)** On the one hand, $\mathcal{L}_B$(get,tell) $\leqslant$ $\mathcal{L}_{MR}$(get,tell) holds by proposition 15. **(ii)** On the other hand, $\mathcal{L}_{MR}$(get,tell) $\not\leqslant$ $\mathcal{L}_B$(get,tell) may be proved as in [BJ99, BJ03b], following the technique used for $\mathcal{L}_{MR}$(ask,tell) $\not\leqslant$ $\mathcal{L}_B$(ask,tell) in part (ii) of proposition 17. This amounts to considering some of the $ask(u_i)$ to be $get(u_i)$ but does not affect the proof further. Intuitively, $\mathcal{L}_B$(get,tell) is unable to atomically retrieve a and b. Let us thus consider $AB = (\{+a, +b\}, \{-a, -b\})$ and assume that $\mathcal{C}(AB)$ is in normal form (see definition 9) and thus is written as $tell(t_1); A_1 + \cdots + tell(t_p); A_p + get(u_1); B_1 + \cdots + get(u_q); B_q$.

The proof proceeds as explain in proof technique 1 by establishing (I) that there is no alternative guarded by a $tell(t_i)$ operation, and (II) that there is no alternative guarded by a $get(u_j)$ operation, in which case, $\mathcal{C}(AB)$ is equivalent to an empty statement, which is not possible since it should contain at least one primitive.

STEP I: Let us first establish that there is no existence of an alternative guarded by a $tell(t_i)$ operation. Otherwise it would point out a failing computation for $\mathcal{C}(AB + (\{\}, \{+a\}))$, contradicting the fact that $\mathcal{O}(AB + (\{\}, \{+a\})) = (\{a\}, \delta^+)$.

STEP II: Let us now establish that there is no alternative guarded by a $get(u_j)$ operation. To that end, let us first consider two auxiliary computations: as $\mathcal{O}((\{\}, \{+a\})) = \{(\{a\}, \delta^+)\}$, any computation of $\mathcal{C}((\{\}, \{+a\}))$ starting in the empty store succeeds. Let $\langle \mathcal{C}((\{\}, \{+a\}))|\varnothing\rangle \rightarrow \cdots \rightarrow \langle E|\{a_1, \ldots, a_m\}\rangle$ be such a computation. Similarly, let $\langle \mathcal{C}((\{\}, \{+b\}))|\varnothing\rangle \rightarrow \cdots \rightarrow \langle E|\{b_1, \ldots, b_n\}\rangle$ be one computation of $\mathcal{C}((\{\}, \{+b\}))$. The proof of the claim proceeds by establishing that none of the u_i's belong to $\{a_1, \ldots, a_m\} \cup \{b_1, \ldots, b_n\}$, in which case a contradiction occurs from the analysis of $\mathcal{C}((\{\}, \{+a\}); (\{\}, \{+b\}); AB)$. As a result, none of the u_i's exist, namely there is no alternative guarded by a $get(u_j)$ operation. □

Proposition 31. $\mathcal{L}_{MR}$(get,tell) $\wr$ $\mathcal{L}_B$(nask,tell)

Proof. **(i)** On the one hand, $\mathcal{L}_{MR}$(get,tell) $\not\leqslant$ $\mathcal{L}_B$(nask,tell). Otherwise, by pattern 3 on non embedding by transitivity, $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_{MR}$(nask,tell) which has been proved impossible in [BJ03b]. **(ii)** On the other hand, $\mathcal{L}_B$(nask,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(get,tell) is established by contradiction, by assuming the existence of a coder $\mathcal{C}$ and by considering $tell(t)$; $nask(t)$. One has $\mathcal{O}(tell(t)$; $nask(t)) = \{(\{t\}, \delta^-)\}$ whereas we shall establish that $\mathcal{C}(tell(t))$; $\mathcal{C}(nask(t))$ has a successful computation. This demonstration is similar to the one used in part (ii) of the proof of proposition 4 of section A.1. □

Proposition 32. $\mathcal{L}_{MR}$(get,tell) $\wr$ $\mathcal{L}_B$(nask,get,tell)

Proof. **(i)** On the one hand, $\mathcal{L}_{MR}$(get,tell) $\not\leqslant$ $\mathcal{L}_B$(nask,get,tell). Otherwise, by pattern 3 of non embedding by transitivity, as $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_{MR}$(get,tell), we then have $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_B$(nask,get,tell) which has been proved impossible in proposition 23. **(ii)** On the other hand, $\mathcal{L}_B$(nask,get,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(get,tell). Otherwise, by pattern 3 we would have $\mathcal{L}_B$(nask,tell) $\leqslant$ $\mathcal{L}_{MR}$(get,tell) which has been proved impossible in proposition 31. □

Proposition 33. $\mathcal{L}_{MR}$(get,tell) $\wr$ $\mathcal{L}_B$(ask,nask,tell)

Proof. **(i)** On the one hand, $\mathcal{L}_{MR}$(get,tell) $\not\leqslant$ $\mathcal{L}_B$(ask,nask,tell). Otherwise, by pattern 3, $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_{MR}$(get,tell) $\leqslant$ $\mathcal{L}_B$(ask,nask,tell) which has been proved impossible in proposition 24(i). **(ii)** On the other hand, $\mathcal{L}_B$(ask,nask,tell) $\not\leqslant$ $\mathcal{L}_{MR}$(get,tell). Otherwise, by pattern 3, one would have $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_{MR}$(ask,nask,tell) $\leqslant$ $\mathcal{L}_B$(get,tell) which has been proved impossible in proposition 22. □

A.2.6 Checking fo presence and/or absence when adding and/or retrieving tokens

Proposition 34. $\mathcal{L}_B$(ask,nask,get,tell) $< \mathcal{L}_{MR}$(ask,nask,get,tell)

Proof. **(i)** On the one hand, $\mathcal{L}_B$(ask,nask,get,tell) $\leqslant \mathcal{L}_{MR}$(ask,nask,get,tell) is immediate by considering the following coder:

$$\begin{aligned} \mathcal{C}(ask(t)) &= (\{+t\},\{\}) \\ \mathcal{C}(get(t)) &= (\{+t\},\{-t\}) \\ \mathcal{C}(nask(t)) &= (\{-t\},\{\}) \\ \mathcal{C}(tell(t)) &= (\{\},\{+t\}) \end{aligned}$$

(ii) On the other hand, $\mathcal{L}_{MR}$(ask,nask,get,tell) $\not\leqslant \mathcal{L}_B$(ask,nask,get,tell) is established by contradiction, using pattern 3 of non embedding by transitivity. Indeed, assuming that $\mathcal{L}_{MR}$(ask,nask,get,tell) $\leqslant \mathcal{L}_B$(ask,nask,get,tell), as $\mathcal{L}_B$(ask,nask,get,tell) $= \mathcal{L}_B$(nask,get,tell), one would have $\mathcal{L}_{MR}$(nask, tell) $\leqslant \mathcal{L}_{MR}$(ask,nask,get,tell) $\leqslant \mathcal{L}_B$(ask,nask,get,tell) $\leqslant \mathcal{L}_B$(nask,get,tell) which has been proved impossible in proposition 26. □

Appendix B

The BachT Language

B.1 The interpreter

B.1.1 The bacht-cli.scala file

This first appendix lists the full code of the BachT interpreter. This one concatenates the abstract class `Expr`, the `BachTParsers` class, the `BachTStore` class and finally the `BachTSimul` class.

```
class Expr
case class bacht_ast_empty_agent() extends Expr
case class bacht_ast_primitive(primitive: String, token: String) extends Expr
case class bacht_ast_agent(op: String, agenti: Expr, agentii: Expr) extends Expr
import scala.util.parsing.combinator._
import scala.util.matching.Regex

class BachTParsers extends RegexParsers {

  def token      : Parser[String] = ("[a-z][0-9a-zA-Z_]*").r ^^ {_.toString}

  val opChoice   : Parser[String] = "+"
  val opPara     : Parser[String] = "||"
  val opSeq      : Parser[String] = ";"

  def primitive : Parser[Expr]   = "tell("~token~")" ^^ {
        case _ ~ vtoken ~ _ => bacht_ast_primitive("tell",vtoken) }  |
                               "ask("~token~")" ^^ {
        case _ ~ vtoken ~ _ => bacht_ast_primitive("ask",vtoken) }   |
                               "get("~token~")" ^^ {
        case _ ~ vtoken ~ _ => bacht_ast_primitive("get",vtoken) }   |
                               "nask("~token~")" ^^ {
        case _ ~ vtoken ~ _ => bacht_ast_primitive("nask",vtoken) }

  def agent = compositionChoice

  def compositionChoice : Parser[Expr] =
```

```
                        compositionPara~rep(opChoice~compositionChoice) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii) => bacht_ast_agent(op,agi,agii) }

  def compositionPara : Parser[Expr] =
                        compositionSeq~rep(opPara~compositionPara) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii) => bacht_ast_agent(op,agi,agii) }

  def compositionSeq : Parser[Expr] = simpleAgent~rep(opSeq~compositionSeq) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii) => bacht_ast_agent(op,agi,agii) }

  def simpleAgent : Parser[Expr] = primitive | parenthesizedAgent

  def parenthesizedAgent : Parser[Expr] = "("~>agent<~")"

}

object BachTSimulParser extends BachTParsers {

  def parse_primitive(prim: String) = parseAll(primitive,prim) match {
        case Success(result, _) => result
        case failure : NoSuccess => scala.sys.error(failure.msg)
  }

  def parse_agent(ag: String) = parseAll(agent,ag) match {
        case Success(result, _) => result
        case failure : NoSuccess => scala.sys.error(failure.msg)
  }

}

import scala.collection.mutable.Map
import scala.swing._

class BachTStore {

   var theStore = Map[String,Int]()

   def tell(token:String):Boolean = {
      if (theStore.contains(token))
        { theStore(token) = theStore(token) + 1 }
      else
        { theStore = theStore ++ Map(token -> 1) }
      true
   }


   def ask(token:String):Boolean = {
      if (theStore.contains(token))
            if (theStore(token) >= 1) { true }
            else { false }
      else false
   }
```

```
   def get(token:String):Boolean = {
      if (theStore.contains(token))
            if (theStore(token) >= 1)
              { theStore(token) = theStore(token) - 1
                true
              }
            else { false }
      else false
   }

   def nask(token:String):Boolean = {
      if (theStore.contains(token))
            if (theStore(token) >= 1) { false }
            else { true }
      else true
   }

   def print_store {
      print("{ ")
      for ((t,d) <- theStore)
         print ( t + "(" + theStore(t) + ")" )
      println(" }")
   }

   def clear_store {
      theStore = Map[String,Int]()
   }

}

object bb extends BachTStore {

   def reset { clear_store }

}

import scala.util.Random
import language.postfixOps

class BachTSimul(var bb: BachTStore) {

   val bacht_random_choice = new Random()

   def run_one(agent: Expr):(Boolean,Expr) = {

      agent match {
         case bacht_ast_primitive(prim,token) =>
            {  if (exec_primitive(prim,token)) { (true,bacht_ast_empty_agent()) }
               else { (false,agent) }
            }

         case bacht_ast_agent(";",ag_i,ag_ii) =>
            {  run_one(ag_i) match
                  { case (false,_) => (false,agent)
```

```
            case (true,bacht_ast_empty_agent()) => (true,ag_ii)
            case (true,ag_cont) => (true,bacht_ast_agent(";",ag_cont,ag_ii))
          }
    }

case bacht_ast_agent("||",ag_i,ag_ii) =>
    {  var branch_choice = bacht_random_choice.nextInt(2)
       if (branch_choice == 0)
         { run_one( ag_i ) match
             { case (false,_) =>
                 { run_one( ag_ii ) match
                     { case (false,_)
                         => (false,agent)
                       case (true,bacht_ast_empty_agent())
                         => (true,ag_i)
                       case (true,ag_cont)
                         => (true,bacht_ast_agent("||",ag_i,ag_cont))
                     }
                 }
               case (true,bacht_ast_empty_agent())
                         => (true,ag_ii)
               case (true,ag_cont)
                         => (true,bacht_ast_agent("||",ag_cont,ag_ii))
             }
         }
       else
         { run_one( ag_ii ) match
             { case (false,_) =>
                 { run_one( ag_i ) match
                     { case (false,_) => (false,agent)
                       case (true,bacht_ast_empty_agent()) => (true,ag_ii)
                       case (true,ag_cont)
                             => (true,bacht_ast_agent("||",ag_cont,ag_ii))
                     }
                 }
               case (true,bacht_ast_empty_agent())
                 => (true,ag_i)
               case (true,ag_cont)
                 => (true,bacht_ast_agent("||",ag_i,ag_cont))
             }
         }
    }

case bacht_ast_agent("+",ag_i,ag_ii) =>
    {  var branch_choice = bacht_random_choice.nextInt(2)
       if (branch_choice == 0)
         { run_one( ag_i ) match
             { case (false,_) =>
                 { run_one( ag_ii ) match
                     { case (false,_) => (false,agent)
                       case (true,bacht_ast_empty_agent())
                         => (true,bacht_ast_empty_agent())
                       case (true,ag_cont)
                         => (true,ag_cont)
                     }
```

```
                    }
                  case (true,bacht_ast_empty_agent())
                    => (true,bacht_ast_empty_agent())
                  case (true,ag_cont)
                    => (true,ag_cont)
                }
            }

          else
            { run_one( ag_ii ) match
                { case (false,_) =>
                    { run_one( ag_i ) match
                        { case (false,_)
                            => (false,agent)
                          case (true,bacht_ast_empty_agent())
                            => (true,bacht_ast_empty_agent())
                          case (true,ag_cont)
                            => (true,ag_cont)
                        }
                    }
                  case (true,bacht_ast_empty_agent())
                    => (true,bacht_ast_empty_agent())
                  case (true,ag_cont)
                    => (true,ag_cont)
                }
            }
        }
    }
}

def bacht_exec_all(agent: Expr):Boolean = {

    var failure = false
    var c_agent = agent
    while ( c_agent != bacht_ast_empty_agent() && !failure ) {
       failure = run_one(c_agent) match
            { case (false,_)          => true
              case (true,new_agent)  =>
                 { c_agent = new_agent
                   false
                 }
            }
        bb.print_store
        println("\n")
    }
    if (c_agent == bacht_ast_empty_agent()) {
        println("Success\n")
        true
    }
    else {
        println("failure\n")
        false
    }
}
```

```
    def exec_primitive(prim:String,token:String):Boolean = {
        prim match
          { case "tell" => bb.tell(token)
            case "ask"  => bb.ask(token)
            case "get"  => bb.get(token)
            case "nask" => bb.nask(token)
          }
    }
}

object ag extends BachTSimul(bb) {

  def apply(agent: String) {
     val agent_parsed = BachTSimulParser.parse_agent(agent)
     ag.bacht_exec_all(agent_parsed)
  }
  def eval(agent:String) { apply(agent) }
  def run(agent:String) { apply(agent) }
}
```

B.2 The command line simulator

B.2.1 The parser

```
class BachTParsers extends RegexParsers {

 def token       : Parser[String] = ("[a-z][0-9a-zA-Z_]*").r ^^ {_.toString}

 val opChoice   : Parser[String] = "+"
 val opPara     : Parser[String] = "||"
 val opSeq      : Parser[String] = ";"

  def primitive : Parser[Expr]   = "tell("~token~")" ^^ {
        case _ ~ vtoken ~ _ => B_AST_Primitive("tell",vtoken) }   |
                                  "ask("~token~")" ^^ {
        case _ ~ vtoken ~ _ => B_AST_Primitive("ask",vtoken) }    |
                                  "get("~token~")" ^^ {
        case _ ~ vtoken ~ _ => B_AST_Primitive("get",vtoken) }    |
                                  "nask("~token~")" ^^ {
        case _ ~ vtoken ~ _ => B_AST_Primitive("nask",vtoken) }

  def agent = compositionChoice

  def compositionChoice : Parser[Expr] =
                                compositionPara~rep(opChoice~compositionChoice) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => B_AST_Agent(op,agi,agii) }

  def compositionPara : Parser[Expr] =
                                 compositionSeq~rep(opPara~compositionPara) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => B_AST_Agent(op,agi,agii) }
```

```
  def compositionSeq : Parser[Expr] =
                                simpleAgent~rep(opSeq~compositionSeq) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => B_AST_Agent(op,agi,agii) }

  def simpleAgent : Parser[Expr] = primitive | parenthesizedAgent

  def parenthesizedAgent : Parser[Expr] = "("~>agent<~")"

}

object BachTSimulParser extends BachTParsers {

  def parse_primitive(prim: String) = parseAll(primitive,prim) match {
        case Success(result, _) => result
        case failure : NoSuccess => scala.sys.error(failure.msg)
  }

  def parse_agent(ag: String) = parseAll(agent,ag) match {
        case Success(result, _) => result
        case failure : NoSuccess => scala.sys.error(failure.msg)
  }

}
```

B.2.2 Vector of continuations

```
def vect_ag_first_steps(lst : List[(Expr,Expr)]) : Vector[Expr] = {
        lst match {
            case Nil => Vector.empty
            case el::lst_rem => el._2 +: vect_ag_first_steps(lst_rem)
        }
}
def lindex_ag_first_steps(lst : List[(Expr,Expr)], i : Int) : List[(Expr,Int)] = {
        var j = i
        lst match {
            case Nil => Nil
            case el::lst_rem => (el._1,j)::lindex_ag_first_steps(lst_rem, j + 1)
        }
}
```

B.2.3 Complete code of the command line simulator

```
/*  BachT command line simulator in command line                          */
/*  - compile file command : scalac codeCompilBachT.scala  */
/*  - execute object       : scala MYSimulator             */

import scala.tools.nsc.Settings
import scala.tools.nsc.interpreter.ILoop

import scala.util.parsing.combinator._
import scala.util.matching.Regex

import scala.util.Random
```

```
import scala.collection.mutable.Map
import scala.collection.mutable.ArrayBuffer
import scala.collection.immutable.Vector
import scala.swing._

import scala.io.StdIn.readLine;

/* Abstract class for BachT */
class Expr
case class B_AST_Empty_Agent() extends Expr
case class B_AST_Primitive(primitive: String, token: String) extends Expr
case class B_AST_Agent(op: String, primitive: Expr, agent: Expr) extends Expr

/* BachT parser : class BachTParsers */

class BachTParsers extends RegexParsers {

  def token      : Parser[String] = ("[a-z][0-9a-zA-Z_]*").r ^^ {_.toString}

  val opChoice   : Parser[String] = "+"
  val opPara     : Parser[String] = "||"
  val opSeq      : Parser[String] = ";"

  def primitive : Parser[Expr]   = "tell("~token~")" ^^ {
        case _ ~ vtoken ~ _ => B_AST_Primitive("tell",vtoken) }  |
                               "ask("~token~")" ^^ {
        case _ ~ vtoken ~ _ => B_AST_Primitive("ask",vtoken) }   |
                               "get("~token~")" ^^ {
        case _ ~ vtoken ~ _ => B_AST_Primitive("get",vtoken) }   |
                               "nask("~token~")" ^^ {
        case _ ~ vtoken ~ _ => B_AST_Primitive("nask",vtoken) }

  def agent = compositionChoice

  def compositionChoice : Parser[Expr] =
                            compositionPara~rep(opChoice~compositionChoice) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => B_AST_Agent(op,agi,agii) }

  def compositionPara : Parser[Expr] =
                                  compositionSeq~rep(opPara~compositionPara) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => B_AST_Agent(op,agi,agii) }

  def compositionSeq : Parser[Expr] = simpleAgent~rep(opSeq~compositionSeq) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => B_AST_Agent(op,agi,agii) }

  def simpleAgent : Parser[Expr] = primitive | parenthesizedAgent

  def parenthesizedAgent : Parser[Expr] = "("~>agent<~")"

}

/* Object BachTSimulParser */
```

```
object BachTSimulParser extends BachTParsers {

  def parse_primitive(prim: String) = parseAll(primitive,prim) match {
        case Success(result, _) => result
        case failure : NoSuccess => scala.sys.error(failure.msg)
  }

  def parse_agent(ag: String) = parseAll(agent,ag) match {
        case Success(result, _) => result
        case failure : NoSuccess => scala.sys.error(failure.msg)
  }

}

/* Object bb representing the store */

object bb {

        var mapTok = Map[String,Int]()

        // synchronized methods for tell, get, ask and nask for a primitive

        def tell(str : String) = bb.synchronized {
                if(mapTok contains str) {
                        mapTok(str) = mapTok(str) + 1
                } else {
                        mapTok = mapTok ++ Map(str -> 1)
                }
                println(">> tell("+str+") successfully terminated")
                print("        >> store :")
                print_store
                println()
                print("BachT> ")
                bb.notifyAll()
                true
        }

        def get(str : String) = bb.synchronized {
                while(!(mapTok contains str) || mapTok(str) == 0) {bb.wait()}
                println(">> get("+str+") successfully terminated")
                mapTok(str) = mapTok(str) - 1
                print("        >> store :")
                print_store
                println()
                print("BachT> ")
                bb.notifyAll()
                true
        }

        def ask(str : String) = bb.synchronized {
                while(!(mapTok contains str) || mapTok(str) == 0) {bb.wait()}
                println(">> ask("+str+") successfully terminated")
                print("        >> store :")
                print_store
```

```
        println()
        print("BachT> ")
        true
}

def nask(str : String) = bb.synchronized {
        while((mapTok contains str)) {bb.wait()}
        if(!(mapTok contains str) || mapTok(str) == 0) {
                println(">> nask("+str+") successfully terminated")
                println("        >> " + "token not present ")
                print("        >> store :")
                print_store
                println()
                print("BachT> ")
        }
        true
}

// test tell, get, ask and nask for execution of first step in a choice
// boolean test if first step is executable

def test_tell(str : String) : Boolean = {
        if(mapTok contains str) {
                mapTok(str) = mapTok(str) + 1
        } else {
                mapTok = mapTok ++ Map(str -> 1)
        }
        println(">> tell("+str+") successfully terminated")
        print("        >> store :")
        print_store
        println()
        print("BachT> ")
        //bb.notifyAll()
        true
}

def test_get(str : String) : Boolean = {
        if(!(mapTok contains str) || mapTok(str) == 0) {return false} else {
        println(">> get("+str+") successfully terminated")
        mapTok(str) = mapTok(str) - 1
        print("        >> store :")
        print_store
        println()
        print("BachT> ")
        //bb.notifyAll()
        return true }
}

def test_ask(str : String) : Boolean = {
        if(!(mapTok contains str) || mapTok(str) == 0) {return false} else {
        println(">> ask("+str+") successfully terminated")
        print("        >> store :")
        print_store
        println()
        print("BachT> ")
```

```
        return true }
}

def test_nask(str : String) : Boolean = {
        if((mapTok contains str)) {return false} else {
        if(!(mapTok contains str) || mapTok(str) == 0) {
                println(">> nask("+str+") successfully terminated")
                println("        >> " + "token not present ")
                print("        >> store :")
                print_store
                println()
                print("BachT> ")
        }
        return true }
}


// Boolean function for execution of previous BachT primitives

def exec_b_prim(prim : Expr) : Boolean = {
        prim match {
                case B_AST_Primitive(b_prim,token) => {
                        b_prim match {
                                case "tell" => {
                                        test_tell(token)
                                }
                                case "get" => {
                                        test_get(token)
                                }
                                case "ask" => {
                                        test_ask(token)
                                }
                                case "nask" => {
                                        test_nask(token)
                                }
                        }
                }
        }
}

// find an executable primitive in the vector of choices

def l_choice(lstEI : List[(Expr,Int)]) : (Boolean,Int) = {
        var found : Boolean = false
        var i : Int = -1
        var lst = lstEI
        while( !found && !lst.isEmpty) {
                if (exec_b_prim((lst.head)._1)) {
                        found = true
                        i = (lst.head)._2
                } else {
                        lst = lst.tail
                }
        }
        return (found,i)
```

```
        }

        // run the list of choices

        def run_l_choice(lst : List[(Expr,Int)]) : Int = bb.synchronized {
                var r = l_choice(lst)
                while(! r._1 ) { // if no executable first step found in the list
                        bb.wait() // wait
                        r = l_choice(lst) // after a notify, restart the search in the list
                }
                return r._2 // if first step found, return the associate integer
        }

        // printing of the store content

        def print_store {
                print("{")
                for ((t,d) <- mapTok)
                        print ( t + "(" + mapTok(t) + ") " )
                println("}")
        }

        // resetting of the store content to 0

        def clear_store {
                mapTok = Map[String,Int]()
                print_store
        }

}

/* class B_Exec for the execution of a BachT agent */

class B_Exec(var current_agent : Expr) {

// function for invoking thread

def thread(body: => Unit): Thread = {
     val t = new Thread {
        override def run() = body
     }
     t.start
     t
}

// function continuation, for operateur op,
// add expr at the end of component el._2 of el
// el._1 contains the first step

def continuation(lst : List[(Expr,Expr)], exp : Expr, op : String)
                                                : List[(Expr,Expr)] = {
        lst match {
                case Nil => Nil
                case el::lst_rem => (el._1,B_AST_Agent(op,el._2,exp))
                                                :: continuation(lst_rem,exp,op)
```

```
        }
}

// function ag_first_steps transforms an agent in a list of
//  pairs Expression-Expression
// The first one is the first step; the second one is its continuation

def ag_first_steps(b_ag : Expr) : List[(Expr,Expr)] = {

        b_ag match {

           // a primitive is the first step followed by the empty agent
           case B_AST_Primitive(b_prim,token) => {
                (B_AST_Primitive(b_prim,token),B_AST_Empty_Agent())::Nil
           }

           // for choice agent, recursive call of the function for
           // every element of the choice
           case B_AST_Agent("+",ag_i,ag_ii) => {
                ag_first_steps(ag_i):::ag_first_steps(ag_ii)
           }

           // a sequence distinguishes both parts of Expression-Expression
           case B_AST_Agent(";",ag_i,ag_ii) => {
                continuation(ag_first_steps(ag_i),ag_ii,";")
           }

           // for parallel agent, ag_i with its continuity in parallel with
           //  ag_ii, and vice versa
           case B_AST_Agent("||",ag_i,ag_ii) => {
                continuation(ag_first_steps(ag_i),ag_ii,"||")
                              :::continuation(ag_first_steps(ag_ii),ag_i,"||")
           }

        }

}

// construction of Vector containing the continuations (el._2)
// after first step (el._1)

def vect_ag_first_steps(lst : List[(Expr,Expr)]) : Vector[Expr] = {
        lst match {
            case Nil => Vector.empty
            case el::lst_rem => el._2 +: vect_ag_first_steps(lst_rem)
        }
}

// Association between first step and index in the Vector

def lindex_ag_first_steps(lst : List[(Expr,Expr)], i : Int)
                                          : List[(Expr,Int)] = {
        var j = i
        lst match {
            case Nil => Nil
```

```
            case el::lst_rem => (el._1,j)
                                ::lindex_ag_first_steps(lst_rem, j + 1)
        }
}

// exec of Dense Bach primitives; invokation of functions of bb

def exec_primitive(b_prim : String, token : String) = {
        b_prim match
                { case "tell" => bb.tell(token)
                  case "ask"  => bb.ask(token)
                  case "get"  => bb.get(token)
                  case "nask" => bb.nask(token)
                }
}

def exec_l_choice(lst : List[(Expr,Int)]) : Int = {
        bb.run_l_choice(lst)
}

// exec agent

def exec(b_ag_parsed : Expr) : Boolean = {
        b_ag_parsed match {
                case B_AST_Empty_Agent() => {true}
                case B_AST_Primitive(b_prim,token) => {
                        exec_primitive(b_prim,token);
                         }
                case B_AST_Agent(";",ag_i,ag_ii)
                        => { if (exec(ag_i)) { exec(ag_ii) } else { false} }
                case B_AST_Agent("||",ag_i,ag_ii) => {
                        val t1 = thread(exec(ag_i))
                        val t2 = thread(exec(ag_ii))
                        t1.join
                        t2.join
                        true
                }
                case B_AST_Agent("+",ag_i,ag_ii) => {
                        var lstEE = ag_first_steps(B_AST_Agent("+",ag_i,ag_ii))
                        var lstEV = vect_ag_first_steps(lstEE)
                        var lstEI = lindex_ag_first_steps(lstEE,0)
                        var i = exec_l_choice(Random.shuffle(lstEI))
                        exec(lstEV(i))
                        true
                }
        }
   }

// generalized exec

def exec_gen(b_ag_parsed : Expr,cpt : Int) = {
   exec(b_ag_parsed)
   bb.synchronized{
        println(">>_Request_" + cpt + "_successfully_terminated")
        println()
```

```
        print("BachT> ")
    }
  }

}

/* Object MYSimInLine containing the main method of the command line simulator */

object MYSimInLine {

   def main(args: Array[String]) {

        println()
        print(" Welcome to BachT version 1.\n")
        print(" Type in agents to evaluate them.\n\n")
        print("BachT> ")

        var myag_parsed : Expr = B_AST_Empty_Agent()
        var line = readLine()
        var cpt : Int = 1

        val maxHist = 5
        var history = ArrayBuffer("m0", "m1", "m2", "m3", "m4")
        var chist = 0
        var ca : String = " "

        while(line != "halt.") {
                line match {
                        case "history." => { bb.synchronized{
                          println("history");
                    println()
             println("   " + "  !: " + history((maxHist+chist-1) % maxHist))
                    println("   " + " !! : " + history((maxHist+chist-2) % maxHist))
                    println("   " + "!!! : " + history((maxHist+chist-3) % maxHist))
                        println("   " + " !v : " + history((maxHist+chist-4) % maxHist))
                        println("   " + "  v : " + history((maxHist+chist-5) % maxHist))
                        println()
                    print("BachT> ")
             }
             line = readLine()
                                        }
                     case "!" => { bb.synchronized{
                                 println()
                                        print("BachT> history mode : ")
                                        println(history((chist-1) % maxHist))
                                        print("BachT>> execute (y/n) : ")
                                        ca = readLine()
                                        if (ca == "y") {
                                              line = history((chist-1) % maxHist) + "."
                                              println()
                                        } else {
                                              println()
                                              print("BachT> ")
                                        }
                                    }
```

```
            if (ca != "y") { line = readLine() }
          }
case "!!" => { bb.synchronized{
            println()
            print("BachT> history mode : ")
            println(history((chist -2) % maxHist))
            print("BachT>> execute (y/n) : ")
            ca = readLine()
            if (ca == "y") {
                line = history((chist -2) % maxHist) + "."
                println()
            } else {
                println()
                print("BachT> ")
            }
          }
          if (ca != "y") { line = readLine() }
        }
case "!!!" => { bb.synchronized{
            println()
            print("BachT> history mode : ")
            println(history((chist -3) % maxHist))
            print("BachT>> execute (y/n) : ")
            ca = readLine()
            if (ca == "y") {
                line = history((chist -3) % maxHist) + "."
                println()
            } else {
                println()
                print("BachT> ")
            }
          }
          if (ca != "y") { line = readLine() }
        }
case "iv" => { bb.synchronized{
            println()
            print("BachT> history mode : ")
            println(history((chist -4) % maxHist))
            print("BachT>> execute (y/n) : ")
            ca = readLine()
            if (ca == "y") {
                line = history((chist -4) % maxHist) + "."
                println()
            } else {
                println()
                print("BachT> ")
            }
          }
          if (ca != "y") { line = readLine() }
        }
case "v" => { bb.synchronized{
            println()
            print("BachT> history mode : ")
            println(history((chist -5) % maxHist))
            print("BachT>> execute (y/n) : ")
```

```
                        ca = readLine()
                        if (ca == "y") {
                            line = history((chist-5) % maxHist) + "."
                            println()
                        } else {
                            println()
                            print("BachT> ")
                        }
                    }
                    if (ca != "y") { line = readLine() }
                }

    case "clear." => {bb.clear_store
                        bb.synchronized{
                            println()
                            print("BachT> ")
                        }
                        line = readLine()
                        }
    case "print." => {bb.print_store
                        bb.synchronized{
                            println()
                            print("BachT> ")
                        }
                        line = readLine()
    }
    case _ => {

            while(!line.endsWith(".")) {
                    line += readLine("   |   ")
            }
            line = line.replace(".","")
            history(chist % maxHist) = line
            chist = (chist + 1)

            try {
                    myag_parsed = BachTSimulParser.parse_agent(line)
                    val mysimul = new B_Exec(myag_parsed)
                    println("BachT> >> Request " + cpt + " launched")
val t = mysimul.thread(mysimul.exec_gen(myag_parsed,cpt))
                    bb.synchronized{
                            println()
                            print("BachT> ")
                    }
                    line = readLine()
            }
            catch {
                    case unknown : Throwable => {
                            bb.synchronized{
                                    println(">> Error of parsing")
                                    println()
                                    print("BachT> ")
                            }
                            line = readLine()
                    }
```

```
                                    }
                                    cpt = cpt + 1
                                }
                            }
                        }
                    }
                }
```

Appendix C

The Dense Bach Language

C.1 The interpreter

C.1.1 The dbach-cli.scala file

This third appendix lists the full code of the Dense Bach interpreter. This one is constitued of the abstract class `Expr`, the `DenseBachParsers` class, the `DenseBachStore` class and finally the `DenseBachSimul` class.

```
class Expr
case class dbach_ast_empty_agent() extends Expr
case class dbach_ast_primitive(primitive: String, token: String,
                                                density: Int) extends Expr
case class dbach_ast_agent(op: String, agenti: Expr,
                                                agentii: Expr) extends Expr

import scala.util.parsing.combinator._
import scala.util.matching.Regex

class DenseBachParsers extends RegexParsers {

  def token     : Parser[String] = ("[a-z][0-9a-zA-Z_]*").r ^^ {_.toString}
  def density   : Parser[Int] = ("[1-9][0-9]*").r ^^ {_.toInt}

  val opChoice  : Parser[String] = "+"
  val opPara    : Parser[String] = "||"
  val opSeq     : Parser[String] = ";"

  def primitive : Parser[Expr]   = "tell(\"token\"(\"density\"))" ^^ {
        case _ ~ vtoken ~ _ ~ vdensity ~ _
                        => dbach_ast_primitive("tell",vtoken,vdensity) }  |
                                "ask(\"token\"(\"density\"))" ^^ {
        case _ ~ vtoken ~ _ ~ vdensity ~ _
                        => dbach_ast_primitive("ask",vtoken,vdensity) }   |
                                "get(\"token\"(\"density\"))" ^^ {
        case _ ~ vtoken ~ _ ~ vdensity ~ _
```

```
                        => dbach_ast_primitive("get",vtoken,vdensity) }  |
                                    "nask("~token~"("~density~"))" ^^ {
        case _ ~ vtoken ~ _ ~ vdensity ~ _
                        => dbach_ast_primitive("nask",vtoken,vdensity) }

  def agent = compositionChoice

  def compositionChoice : Parser[Expr] =
                        compositionPara~rep(opChoice~compositionChoice) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => dbach_ast_agent(op,agi,agii) }

  def compositionPara : Parser[Expr] =
                        compositionSeq~rep(opPara~compositionPara) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => dbach_ast_agent(op,agi,agii) }

  def compositionSeq : Parser[Expr] =
                             simpleAgent~rep(opSeq~compositionSeq) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => dbach_ast_agent(op,agi,agii) }

  def simpleAgent : Parser[Expr] = primitive | parenthesizedAgent

  def parenthesizedAgent : Parser[Expr] = "("~>agent<~")"
}

object DenseBachSimulParser extends DenseBachParsers {

  def parse_primitive(prim: String) = parseAll(primitive,prim) match {
        case Success(result, _) => result
        case failure : NoSuccess => scala.sys.error(failure.msg)
  }

  def parse_agent(ag: String) = parseAll(agent,ag) match {
        case Success(result, _) => result
        case failure : NoSuccess => scala.sys.error(failure.msg)
  }
}

import scala.collection.mutable.Map
import scala.swing._

class DenseBachStore {

   var theStore = Map[String,Int]()

   def tell(token:String,density:Int):Boolean = {
      if (theStore.contains(token))
        { theStore(token) = theStore(token) + density }
      else
        { theStore = theStore ++ Map(token -> density) }
      true
   }
```

```
    def ask(token:String,density:Int):Boolean = {
       if (theStore.contains(token))
              if (theStore(token) >= density) { true }
              else { false }
       else false
    }

    def get(token:String,density:Int):Boolean = {
       if (theStore.contains(token))
              if (theStore(token) >= density)
                { theStore(token) = theStore(token) - density
                  true
                }
              else { false }
       else false
    }

    def nask(token:String,density:Int):Boolean = {
       if (theStore.contains(token))
              if (theStore(token) >= density) { false }
              else { true }
       else true
    }

    def print_store {
       print("{ ")
       for ((t,d) <- theStore)
          print ( t + "(" + theStore(t) + ")" )
       println(" }")
    }

    def clear_store {
       theStore = Map[String,Int]()
    }

}

object bb extends DenseBachStore {

    def reset { clear_store }

}
import scala.util.Random
import language.postfixOps

class DenseBachSimul(var bb: DenseBachStore) {

    val dbach_random_choice = new Random()

    def run_one(agent: Expr):(Boolean,Expr) = {

       agent match {
          case dbach_ast_primitive(prim,token) =>
            { if (exec_primitive(prim,token)) { (true,dbach_ast_empty_agent()) }
              else { (false,agent) }
```

```
  }

case dbach_ast_agent(";",ag_i,ag_ii) =>
  { run_one(ag_i) match
     { case (false,_) => (false,agent)
       case (true,dbach_ast_empty_agent()) => (true,ag_ii)
       case (true,ag_cont) => (true,dbach_ast_agent(";",ag_cont,ag_ii))
     }
  }

case dbach_ast_agent("||",ag_i,ag_ii) =>
  { var branch_choice = dbach_random_choice.nextInt(2)
    if (branch_choice == 0)
      { run_one( ag_i ) match
         { case (false,_) =>
            { run_one( ag_ii ) match
               { case (false,_)
                        => (false,agent)
                 case (true,dbach_ast_empty_agent())
                        => (true,ag_i)
                 case (true,ag_cont)
                        => (true,dbach_ast_agent("||",ag_i,ag_cont))
               }
            }
           case (true,dbach_ast_empty_agent())
                        => (true,ag_ii)
           case (true,ag_cont)
                        => (true,dbach_ast_agent("||",ag_cont,ag_ii))
         }
      }
    else
      { run_one( ag_ii ) match
         { case (false,_) =>
            { run_one( ag_i ) match
               { case (false,_) => (false,agent)
                 case (true,dbach_ast_empty_agent()) => (true,ag_ii)
                 case (true,ag_cont)
                        => (true,dbach_ast_agent("||",ag_cont,ag_ii))
               }
            }
           case (true,dbach_ast_empty_agent()) => (true,ag_i)
           case (true,ag_cont)
                         => (true,dbach_ast_agent("||",ag_i,ag_cont))
         }
      }
  }

case dbach_ast_agent("+",ag_i,ag_ii) =>
   {  var branch_choice = dbach_random_choice.nextInt(2)
      if (branch_choice == 0)
        { run_one( ag_i ) match
            { case (false,_) =>
               { run_one( ag_ii ) match
                  { case (false,_) => (false,agent)
                    case (true,dbach_ast_empty_agent())
```

```
                                => (true,dbach_ast_empty_agent())
                              case (true,ag_cont)
                                => (true,ag_cont)
                            }
                        }
                      case (true,dbach_ast_empty_agent())
                        => (true,dbach_ast_empty_agent())
                      case (true,ag_cont)
                        => (true,ag_cont)
                    }
                }
              else
                { run_one( ag_ii ) match
                    { case (false,_) =>
                        { run_one( ag_i ) match
                            { case (false,_)
                                => (false,agent)
                              case (true,dbach_ast_empty_agent())
                                => (true,dbach_ast_empty_agent())
                              case (true,ag_cont)
                                => (true,ag_cont)
                            }
                        }
                      case (true,dbach_ast_empty_agent())
                        => (true,dbach_ast_empty_agent())
                      case (true,ag_cont)
                        => (true,ag_cont)
                    }
                }
            }
    }
 }

def dbach_exec_all(agent: Expr):Boolean = {

     var failure = false
     var c_agent = agent
     while ( c_agent != dbach_ast_empty_agent() && !failure ) {
        failure = run_one(c_agent) match
            { case (false,_)          => true
              case (true,new_agent)  =>
                 { c_agent = new_agent
                   false
                 }
            }
         bb.print_store
         println("\n")
     }

     if (c_agent == dbach_ast_empty_agent()) {
         println("Success\n")
         true
     }
     else {
         println("failure\n")
```

```
            false
        }
    }

    def exec_primitive(prim:String,token:String,density:Int):Boolean = {
        prim match
          { case "tell" => bb.tell(token,density)
            case "ask"  => bb.ask(token,density)
            case "get"  => bb.get(token,density)
            case "nask" => bb.nask(token,density)
          }
    }
}

object ag extends DenseBachSimul(bb) {

  def apply(agent: String) {
     val agent_parsed = DenseBachSimulParser.parse_agent(agent)
     ag.dbach_exec_all(agent_parsed)
  }
  def eval(agent:String) { apply(agent) }
  def run(agent:String) { apply(agent) }

}
```

C.2 The command line simulator

C.2.1 Abstract class

```
/*  Dense Bach simulator in command line                  */
/*  - compile file command : scalac codeCompilV2.scala  */
/*  - execute object       : scala MYSimulator          */

import scala.tools.nsc.Settings
import scala.tools.nsc.interpreter.ILoop

import scala.util.parsing.combinator._
import scala.util.matching.Regex

import scala.util.Random
import scala.collection.mutable.Map
import scala.collection.mutable.ArrayBuffer
import scala.collection.immutable.Vector
import scala.swing._

import scala.io.StdIn.readLine;

/* Abstract class for Dense Bach */
class Expr
case class DB_AST_Empty_Agent() extends Expr
case class DB_AST_Primitive(primitive: String, token: String, density: Int) extends Expr
case class DB_AST_Agent(op: String, primitive: Expr, agent: Expr) extends Expr
```

C.2.2 Dense Bach Parser

```
/* Dense Bach parser : class DenseBachParsers */

class DenseBachParsers extends RegexParsers {

  def token      : Parser[String] = ("[a-z][0-9a-zA-Z_]*").r ^^ {_.toString}
  def density    : Parser[Int] = ("[1-9][0-9]*").r ^^ {_.toInt}

  val opChoice   : Parser[String] = "+"
  val opPara     : Parser[String] = "||"
  val opSeq      : Parser[String] = ";"

  def primitive : Parser[Expr]   = "tell("~token~"("~density~"))" ^^ {
        case _ ~ vtoken ~ _ ~ vdensity ~ _ => DB_AST_Primitive("tell",vtoken,vdensity) }
|
                                   "ask("~token~"("~density~"))" ^^ {
        case _ ~ vtoken ~ _ ~ vdensity ~ _  => DB_AST_Primitive("ask",vtoken,vdensity) }
|
                                   "get("~token~"("~density~"))" ^^ {
        case _ ~ vtoken ~ _ ~ vdensity ~ _  => DB_AST_Primitive("get",vtoken,vdensity) }
|
                                   "nask("~token~"("~density~"))" ^^ {
        case _ ~ vtoken ~ _ ~ vdensity ~ _  => DB_AST_Primitive("nask",vtoken,vdensity) }

  def agent = compositionChoice

  def compositionChoice : Parser[Expr] = compositionPara~rep(opChoice~compositionChoice) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => DB_AST_Agent(op,agi,agii) }

  def compositionPara : Parser[Expr] = compositionSeq~rep(opPara~compositionPara) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => DB_AST_Agent(op,agi,agii) }

  def compositionSeq : Parser[Expr] = simpleAgent~rep(opSeq~compositionSeq) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => DB_AST_Agent(op,agi,agii) }

  def simpleAgent : Parser[Expr] = primitive | parenthesizedAgent

  def parenthesizedAgent : Parser[Expr] = "("~>agent<~")"

}

/* Object DenseBachSimulParser */

object DenseBachSimulParser extends DenseBachParsers {

  def parse_primitive(prim: String) = parseAll(primitive,prim) match {
        case Success(result, _) => result
        case failure : NoSuccess => scala.sys.error(failure.msg)
  }

  def parse_agent(ag: String) = parseAll(agent,ag) match {
```

```
        case Success(result, _) => result
        case failure : NoSuccess => scala.sys.error(failure.msg)
  }

}
```

C.2.3 The store

```
/* Object bb representing the store */

object bb {

        var mapTok = Map[String,Int]()

        // synchronized methods for tell, get, ask and nask

        def tell(str : String, density : Int) = bb.synchronized {
                if(mapTok contains str) {
                        mapTok(str) = mapTok(str) + density
                } else {
                        mapTok = mapTok ++ Map(str -> density)
                }
                println(">> tell("+str+"("+density+")) successfully terminated")
                print("        >> store :")
                print_store
                println()
                print("DBach> ")
                bb.notifyAll()
                true
        }

        def get(str : String, density : Int) = bb.synchronized {
                while(!(mapTok contains str) || (mapTok(str) < density)) {bb.wait()}
                println(">> get("+str+"("+density+")) successfully terminated")
                mapTok(str) = mapTok(str) - density
                print("        >> store :")
                print_store
                println()
                print("DBach> ")
                true
        }

        def ask(str : String, density : Int) = bb.synchronized {
                while(!(mapTok contains str) || (mapTok(str) < density)) {bb.wait()}
                println(">> ask("+str+"("+density+")) successfully terminated")
                print("        >> store :")
                print_store
                println()
                print("DBach> ")
                true
        }

        def nask(str : String, density : Int) = bb.synchronized {
                while((mapTok contains str) && (mapTok(str) >= density)) {bb.wait()}
                if(!(mapTok contains str)) {
```

```
                println(">> nask("+str+"("+density+")) successfully terminated")
                println("        >> " + "token " + str + " not present ")
                print("        >> store :")
                print_store
                println()
                print("DBach> ")
        } else {
                println(">> nask("+str+"("+density+")) successfully terminated")
                print("        >> store :")
                print_store
                println()
                print("DBach> ")
        }
        true
}

def test_tell(str : String, density : Int) : Boolean = {
        if(mapTok contains str) {
                mapTok(str) = mapTok(str) + density
        } else {
                mapTok = mapTok ++ Map(str -> density)
        }
        println(">> tell("+str+"("+density+")) successfully terminated")
        print("        >> store :")
        print_store
        println()
        print("DBach> ")
        //bb.notifyAll()
        return true
}

def test_get(str : String, density : Int) : Boolean = {
        if(!(mapTok contains str) || (mapTok(str) < density)) { return false } else {
        println(">> get("+str+"("+density+")) successfully terminated")
        mapTok(str) = mapTok(str) - density
        print("        >> store :")
        print_store
        println()
        print("DBach> ")
        return true }
}

def test_ask(str : String, density : Int) : Boolean = {
        if(!(mapTok contains str) || (mapTok(str) < density)) { return false } else {
        println(">> ask("+str+"("+density+")) successfully terminated")
        print("        >> store :")
        print_store
        println()
        print("DBach> ")
        return true }
}

def test_nask(str : String, density : Int) : Boolean = {
        if((mapTok contains str) && (mapTok(str) >= density)) { return false } else {
        if(!(mapTok contains str)) {
```

```
                println(">> nask("+str+"("+density+")) successfully terminated")
                println("        >> " + "token " + str + " not present ")
                print("        >> store :")
                print_store
                println()
                print("DBach> ")
        } else {
                println(">> nask("+str+"("+density+")) successfully terminated")
                print("        >> store :")
                print_store
                println()
                print("DBach> ")
        }
        return true }
}

// Boolean function for execution of previous Dense Bach primitives

def exec_db_prim(prim : Expr) : Boolean = {
        prim match {
                case DB_AST_Primitive(db_prim,token,density) => {
                        db_prim match {
                                case "tell" => {
                                        test_tell(token,density)
                                }
                                case "get" => {
                                        test_get(token,density)
                                }
                                case "ask" => {
                                        test_ask(token,density)
                                }
                                case "nask" => {
                                        test_nask(token,density)
                                }
                        }
                }
        }
}

// find an executable primitive in the vector of choices

def run_l_choice(lstEI : List[(Expr,Int)]) : (Boolean,Int) = {
        var found : Boolean = false
        var i : Int = -1
        var lst = lstEI
        while( !found && !lst.isEmpty) {
                if (exec_db_prim((lst.head)._1)) {
                        found = true
                        i = (lst.head)._2
                } else {
                        lst = lst.tail
                }
        }
        return (found,i)
}
```

```
        // run the list of choices

        def l_choice(lst : List[(Expr,Int)]) : Int = bb.synchronized {
                var r = run_l_choice(lst)
                while(! r._1 ) {
                        bb.wait()
                        r = run_l_choice(lst)
                }
                return r._2
                //notifyAll()
        }

        // printing of the store content

        def print_store {
                print("{")
                for ((t,d) <- mapTok)
                        print ( t + "(" + mapTok(t) + ") " )
                println("}")
        }

        // resetting of the store content to 0

        def clear_store {
                mapTok = Map[String,Int]()
                print_store
        }

}
```

C.2.4 Executing a Dense Bach Agent

```
/* class DB_Exec for the execution of a Dense Bach agent */

class DB_Exec(var current_agent : Expr) {

// function for invoking thread

def thread(body: => Unit): Thread = {
     val t = new Thread {
        override def run() = body
     }
     t.start
     t
}

// function ext_continuation, for operateur op, add expr at the end of component el._2 of el
// el._1 contains the first step

def ext_continuation(lst : List[(Expr,Expr)], exp : Expr, op : String) : List[(Expr,Expr)] = {
        lst match {
     case Nil => Nil
     case el::lst_rem => (el._1,DB_AST_Agent(op,el._2,exp)) :: ext_continuation(lst_rem,exp,op)
        }
```

```
}

// function ag_first_steps transforms an agent in a list of pairs Expression-Expression
// The first one is the first step; the second one is its continuation

def ag_first_steps(db_ag : Expr) : List[(Expr,Expr)] = {

        db_ag match {

           // a primitive is the first step followed by the empty agent
           case DB_AST_Primitive(db_prim,token,density) => {
                (DB_AST_Primitive(db_prim,token,density),DB_AST_Empty_Agent())::Nil
           }

           // for choice agent, recursive call of the function for every element of the choice
           case DB_AST_Agent("+",ag_i,ag_ii) => {
                ag_first_steps(ag_i):::ag_first_steps(ag_ii)
           }

           // a sequence distinguishes both parts of Expression-Expression
           case DB_AST_Agent(";",ag_i,ag_ii) => {
                ext_continuation(ag_first_steps(ag_i),ag_ii,";")
           }

           // for parallel agent, ag_i ith its continuity in parallel with ag_ii, and vice versa
           case DB_AST_Agent("||",ag_i,ag_ii) => {
                ext_continuation(ag_first_steps(ag_i),ag_ii,"||")
                                        :::ext_continuation(ag_first_steps(ag_ii),ag_i,"||")
           }

        }

}

// construction of Vector containing the continuations (el._2) after first step (el._1)

def vect_ag_first_steps(lst : List[(Expr,Expr)]) : Vector[Expr] = {
        lst match {
                case Nil => Vector.empty
                case el::lst_rem => el._2 +: vect_ag_first_steps(lst_rem)
        }
}

// Association between first step and index in the Vector

def lindex_ag_first_steps(lst : List[(Expr,Expr)], i : Int) : List[(Expr,Int)] = {
        var j = i
        lst match {
                case Nil => Nil
                case el::lst_rem => (el._1,j)::lindex_ag_first_steps(lst_rem, j + 1)
        }
}

// exec of Dense Bach primitives; invokation of functions of bb
```

```
def exec_primitive(db_prim : String, token : String, density : Int) = {
        db_prim match
                { case "tell" => bb.tell(token,density)
                  case "ask"  => bb.ask(token,density)
                  case "get"  => bb.get(token,density)
                  case "nask" => bb.nask(token,density)
                }
}

def exec_l_choice(lst : List[(Expr,Int)]) : Int = {
        bb.l_choice(lst)
}

// exec agent

def exec(db_ag_parsed : Expr) : Boolean = {
        db_ag_parsed match {
                case DB_AST_Empty_Agent() => {true}
                case DB_AST_Primitive(db_prim,token,density) => {
                        exec_primitive(db_prim,token,density);
                         }
                case DB_AST_Agent(";",ag_i,ag_ii) => {
                                if (exec(ag_i)) { exec(ag_ii) } else { false} }
                case DB_AST_Agent("||",ag_i,ag_ii) => {
                        val t1 = thread(exec(ag_i))
                        val t2 = thread(exec(ag_ii))
                        t1.join
                        t2.join
                        true
                }
                case DB_AST_Agent("+",ag_i,ag_ii) => {
                        var lstEE = ag_first_steps(DB_AST_Agent("+",ag_i,ag_ii))
                        var lstEV = vect_ag_first_steps(lstEE)
                        var lstEI = lindex_ag_first_steps(lstEE,0)
                        var i = exec_l_choice(Random.shuffle(lstEI))
                        exec(lstEV(i))
                        true
                }
        }
   }

// generalized exec

def exec_gen(db_ag_parsed : Expr,cpt : Int) = {
   exec(db_ag_parsed)
   bb.synchronized{
        println(">> Request " + cpt + " successfully terminated")
        println()
        print("DBach> ")
   }
 }

}
```

C.2.5 The Command Line Simulator

```
/* Object MYSimInLine containing the main method of the command line simulator */

object MYSimInLine {

   def main(args: Array[String]) {

        println()
        print("␣Welcome␣to␣Dense␣Bach␣version␣1.\n")
        print("␣Type␣in␣agents␣to␣evaluate␣them.\n\n")
        print("DBach>␣")

        var myag_parsed : Expr = DB_AST_Empty_Agent()
        var line = readLine()
        var cpt : Int = 1

        val maxHist = 5
        var history = ArrayBuffer("m0", "m1", "m2", "m3", "m4")
        var chist = 0
        var ca : String = "␣"

        while(line != "halt.") {
                line match {
                        case "history." => { bb.synchronized{
              println("history");
              println()
              println("␣␣␣" + "␣␣!␣:" + history((maxHist+chist-1) % maxHist))
              println("␣␣␣" + "␣!!␣:" + history((maxHist+chist-2) % maxHist))
              println("␣␣␣" + "!!!␣:" + history((maxHist+chist-3) % maxHist))
              println("␣␣␣" + "␣!v␣:" + history((maxHist+chist-4) % maxHist))
              println("␣␣␣" + "␣␣v␣:" + history((maxHist+chist-5) % maxHist))
              println()
              print("DBach>␣")
                                  }
                                  line = readLine()
                                     }
                        case "!" => { bb.synchronized{
                                        println()
                                        print("DBach>␣history␣mode␣:␣")
                                        println(history((chist-1) % maxHist))
                                        print("DBach>>␣execute␣(y/n)␣:␣")
                                        ca = readLine()
                                        if (ca == "y") {
                                             line = history((chist-1) % maxHist) + "."
                                             println()
                                        } else {
                                             println()
                                             print("DBach>␣")
                                        }
                                     }
                                     if (ca != "y") { line = readLine() }
                                   }
                        case "!!" => { bb.synchronized{
                                        println()
```

```
                print("DBach> history mode : ")
                println(history((chist-2) % maxHist))
                print("DBach>> execute (y/n) : ")
                ca = readLine()
                if (ca == "y") {
                    line = history((chist-2) % maxHist) + "."
                    println()
                } else {
                    println()
                    print("DBach> ")
                }
            }
            if (ca != "y") { line = readLine() }
        }
    case "!!!" => { bb.synchronized{
                println()
                print("DBach> history mode : ")
                println(history((chist-3) % maxHist))
                print("DBach>> execute (y/n) : ")
                ca = readLine()
                if (ca == "y") {
                    line = history((chist-3) % maxHist) + "."
                    println()
                } else {
                    println()
                    print("DBach> ")
                }
            }
            if (ca != "y") { line = readLine() }
        }
    case "iv" => { bb.synchronized{
                println()
                print("DBach> history mode : ")
                println(history((chist-4) % maxHist))
                print("DBach>> execute (y/n) : ")
                ca = readLine()
                if (ca == "y") {
                    line = history((chist-4) % maxHist) + "."
                    println()
                } else {
                    println()
                    print("DBach> ")
                }
            }
            if (ca != "y") { line = readLine() }
        }
    case "v" => { bb.synchronized{
                println()
                print("DBach> history mode : ")
                println(history((chist-5) % maxHist))
                print("DBach>> execute (y/n) : ")
                ca = readLine()
                if (ca == "y") {
                    line = history((chist-5) % maxHist) + "."
                    println()
```

```
                        } else {
                            println()
                            print("DBach> ")
                        }
                    }
                    if (ca != "y") { line = readLine() }
                }
            case "clear." => {bb.clear_store
                    bb.synchronized{
                        println()
                        print("DBach> ")
                    }
                    line = readLine()
                    }
            case "print." => {bb.print_store
                    bb.synchronized{
                        println()
                        print("DBach> ")
                    }
                    line = readLine()
            }
            case _ => {

                while(!line.endsWith(".")) {
                    line += readLine("    |    ")
                }
                line = line.replace(".","")
                history(chist % maxHist) = line
                chist = (chist + 1)

                try {
                    myag_parsed = DenseBachSimulParser.parse_agent(line)
                    val mysimul = new DB_Exec(myag_parsed)
                    println("DBach> >> Request " + cpt + " launched")
        val t = mysimul.thread(mysimul.exec_gen(myag_parsed,cpt))
                    bb.synchronized{
                        println()
                        print("DBach> ")
                    }
                    line = readLine()
                }
                catch {
                    case unknown : Throwable => {
                        bb.synchronized{
                            println(">> Error of parsing")
                            println()
                            print("DBach> ")
                        }
                        line = readLine()
                    }
                }
                cpt = cpt + 1
            }
        }
}
```

```
    }
}
```

C.2.6 Complete code of the command line simulator

```
/*  Dense Bach simulator in command line                */
/*  - compile file command : scalac codeCompilV2.scala  */
/*  - execute object       : scala MYSimulator          */

import scala.tools.nsc.Settings
import scala.tools.nsc.interpreter.ILoop

import scala.util.parsing.combinator._
import scala.util.matching.Regex

import scala.util.Random
import scala.collection.mutable.Map
import scala.collection.mutable.ArrayBuffer
import scala.collection.immutable.Vector
import scala.swing._

import scala.io.StdIn.readLine;

/* Abstract class for Dense Bach */
class Expr
case class DB_AST_Empty_Agent() extends Expr
case class DB_AST_Primitive(primitive: String, token: String,
                                              density: Int) extends Expr
case class DB_AST_Agent(op: String, primitive: Expr,
                                               agent: Expr) extends Expr

/* Dense Bach parser : class DenseBachParsers */

class DenseBachParsers extends RegexParsers {

  def token : Parser[String] = ("[a-z][0-9a-zA-Z_]*").r ^^ {_.toString}
  def density : Parser[Int] = ("[1-9][0-9]*").r ^^ {_.toInt}

  val opChoice  : Parser[String] = "+"
  val opPara    : Parser[String] = "||"
  val opSeq     : Parser[String] = ";"

  def primitive : Parser[Expr]   = "tell(~token~"(~density~"))" ^^ {
        case _ ~ vtoken ~ _ ~ vdensity ~ _
                    => DB_AST_Primitive("tell",vtoken,vdensity) }  |
                          "ask(~token~"(~density~"))" ^^ {
        case _ ~ vtoken ~ _ ~ vdensity ~ _
                    => DB_AST_Primitive("ask",vtoken,vdensity) }   |
                             "get(~token~"(~density~"))" ^^ {
        case _ ~ vtoken ~ _ ~ vdensity ~ _
                    => DB_AST_Primitive("get",vtoken,vdensity) }   |
                             "nask(~token~"(~density~"))" ^^ {
        case _ ~ vtoken ~ _ ~ vdensity ~ _
                    => DB_AST_Primitive("nask",vtoken,vdensity) }
```

```
  def agent = compositionChoice

  def compositionChoice : Parser[Expr] =
                compositionPara~rep(opChoice~compositionChoice) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => DB_AST_Agent(op,agi,agii) }

  def compositionPara : Parser[Expr] =
                      compositionSeq~rep(opPara~compositionPara) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => DB_AST_Agent(op,agi,agii) }

  def compositionSeq : Parser[Expr] =
                           simpleAgent~rep(opSeq~compositionSeq) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => DB_AST_Agent(op,agi,agii) }

  def simpleAgent : Parser[Expr] = primitive | parenthesizedAgent

  def parenthesizedAgent : Parser[Expr] = "("~>agent<~")"

}

/* Object DenseBachSimulParser */

object DenseBachSimulParser extends DenseBachParsers {

  def parse_primitive(prim: String) = parseAll(primitive,prim) match {
        case Success(result, _) => result
        case failure : NoSuccess => scala.sys.error(failure.msg)
  }

  def parse_agent(ag: String) = parseAll(agent,ag) match {
        case Success(result, _) => result
        case failure : NoSuccess => scala.sys.error(failure.msg)
  }

}

/* Object bb representing the store */

object bb {

        var mapTok = Map[String,Int]()

        // synchronized methods for tell, get, ask and nask

        def tell(str : String, density : Int) = bb.synchronized {
                if(mapTok contains str) {
                        mapTok(str) = mapTok(str) + density
                } else {
                        mapTok = mapTok ++ Map(str -> density)
                }
                println(">> tell("+str+"("+density+")) successfully terminated")
                print("        >> store :")
```

```
        print_store
        println()
        print("DBach> ")
        bb.notifyAll()
        true
}

def get(str : String, density : Int) = bb.synchronized {
        while(!(mapTok contains str) || (mapTok(str) < density)) {bb.wait()}
        println(">> get("+str+"("+density+")) successfully terminated")
        mapTok(str) = mapTok(str) - density
        print("        >> store :")
        print_store
        println()
        print("DBach> ")
        true
}

def ask(str : String, density : Int) = bb.synchronized {
        while(!(mapTok contains str) || (mapTok(str) < density)) {bb.wait()}
        println(">> ask("+str+"("+density+")) successfully terminated")
        print("        >> store :")
        print_store
        println()
        print("DBach> ")
        true
}

def nask(str : String, density : Int) = bb.synchronized {
        while((mapTok contains str) && (mapTok(str) >= density)) {bb.wait()}
        if(!(mapTok contains str)) {
                println(">> nask("+str+"("+density+")) successfully terminated")
                println("        >> " + "token " + str + " not present ")
                print("        >> store :")
                print_store
                println()
                print("DBach> ")
        } else {
                println(">> nask("+str+"("+density+")) successfully terminated")
                print("        >> store :")
                print_store
                println()
                print("DBach> ")
        }
        true
}

def test_tell(str : String, density : Int) : Boolean = {
        if(mapTok contains str) {
                mapTok(str) = mapTok(str) + density
        } else {
                mapTok = mapTok ++ Map(str -> density)
        }
        println(">> tell("+str+"("+density+")) successfully terminated")
        print("        >> store :")
```

```
        print_store
        println()
        print("DBach> ")
        //bb.notifyAll()
        return true
}

def test_get(str : String, density : Int) : Boolean = {
        if(!(mapTok contains str) || (mapTok(str) < density)) { return false }
                else {
        println(">> get("+str+"("+density+")) successfully terminated")
        mapTok(str) = mapTok(str) - density
        print("        >> store :")
        print_store
        println()
        print("DBach> ")
        return true }
}

def test_ask(str : String, density : Int) : Boolean = {
        if(!(mapTok contains str) || (mapTok(str) < density)) { return false }
                else {
        println(">> ask("+str+"("+density+")) successfully terminated")
        print("        >> store :")
        print_store
        println()
        print("DBach> ")
        return true }
}

def test_nask(str : String, density : Int) : Boolean = {
        if((mapTok contains str) && (mapTok(str) >= density)) { return false }
                else {
        if(!(mapTok contains str)) {
                println(">> nask("+str+"("+density+")) successfully terminated")
                println("        >> " + "token " + str + " not present ")
                print("        >> store :")
                print_store
                println()
                print("DBach> ")
        } else {
                println(">> nask("+str+"("+density+")) successfully terminated")
                print("        >> store :")
                print_store
                println()
                print("DBach> ")
        }
        return true }
}

// Boolean function for execution of previous Dense Bach primitives

def exec_db_prim(prim : Expr) : Boolean = {
        prim match {
                case DB_AST_Primitive(db_prim,token,density) => {
```

```
                        db_prim match {
                                case "tell" => {
                                        test_tell(token,density)
                                }
                                case "get" => {
                                        test_get(token,density)
                                }
                                case "ask" => {
                                        test_ask(token,density)
                                }
                                case "nask" => {
                                        test_nask(token,density)
                                }
                        }
                }
        }
}

// find an executable primitive in the vector of choices

def run_l_choice(lstEI : List[(Expr,Int)]) : (Boolean,Int) = {
        var found : Boolean = false
        var i : Int = -1
        var lst = lstEI
        while( !found && !lst.isEmpty) {
                if (exec_db_prim((lst.head)._1)) {
                        found = true
                        i = (lst.head)._2
                } else {
                        lst = lst.tail
                }
        }
        return (found,i)
}

// run the list of choices

def l_choice(lst : List[(Expr,Int)]) : Int = bb.synchronized {
        var r = run_l_choice(lst)
        while(! r._1 ) {
                bb.wait()
                r = run_l_choice(lst)
        }
        return r._2
        //notifyAll()
}

// printing of the store content

def print_store {
        print("_{_")
        for ((t,d) <- mapTok)
                print ( t + "(" + mapTok(t) + ")_" )
        println("}")
}
```

```
        // resetting of the store content to 0

        def clear_store {
                mapTok = Map[String,Int]()
                print_store
        }

}

/* class DB_Exec for the execution of a Dense Bach agent */

class DB_Exec(var current_agent : Expr) {

// function for invoking thread

def thread(body: => Unit): Thread = {
     val t = new Thread {
        override def run() = body
     }
     t.start
     t
}

// function ext_continuation, for operateur op, add expr
// at the end of component el._2 of el
// el._1 contains the first step

def ext_continuation(lst : List[(Expr,Expr)], exp : Expr,
                                op : String) : List[(Expr,Expr)] = {
        lst match {
                case Nil => Nil
                case el::lst_rem => (el._1,DB_AST_Agent(op,el._2,exp))
                                        :: ext_continuation(lst_rem,exp,op)
        }
}

// function ag_first_steps transforms an agent in
// a list of pairs Expression-Expression
// The first one is the first step; the second one is its continuation

def ag_first_steps(db_ag : Expr) : List[(Expr,Expr)] = {

        db_ag match {

           // a primitive is the first step followed by the empty agent
           case DB_AST_Primitive(db_prim,token,density) => {
                (DB_AST_Primitive(db_prim,token,density),DB_AST_Empty_Agent())::Nil
           }

           // for choice agent, recursive call of the function
           // for every element of the choice
           case DB_AST_Agent("+",ag_i,ag_ii) => {
                ag_first_steps(ag_i):::ag_first_steps(ag_ii)
           }
```

```
            // a sequence distinguishes both parts of Expression-Expression
            case DB_AST_Agent(";",ag_i,ag_ii) => {
                 ext_continuation(ag_first_steps(ag_i),ag_ii,";")
            }

            // for parallel agent, ag_i ith its continuity in parallel with
            // ag_ii, and vice versa
            case DB_AST_Agent("||",ag_i,ag_ii) => {
                 ext_continuation(ag_first_steps(ag_i),ag_ii,"||")
                           :::ext_continuation(ag_first_steps(ag_ii),ag_i,"||")
            }

        }

}

// construction of Vector containing the continuations (el._2)
// after first step (el._1)

def vect_ag_first_steps(lst : List[(Expr,Expr)]) : Vector[Expr] = {
        lst match {
                case Nil => Vector.empty
                case el::lst_rem => el._2 +: vect_ag_first_steps(lst_rem)
        }
}

// Association between first step and index in the Vector

def lindex_ag_first_steps(lst : List[(Expr,Expr)], i : Int)
                                              : List[(Expr,Int)] = {
        var j = i
        lst match {
                case Nil => Nil
                case el::lst_rem => (el._1,j)
                                    ::lindex_ag_first_steps(lst_rem, j + 1)
        }
}

// exec of Dense Bach primitives; invokation of functions of bb

def exec_primitive(db_prim : String, token : String, density : Int) = {
        db_prim match
                { case "tell" => bb.tell(token,density)
                  case "ask"  => bb.ask(token,density)
                  case "get"  => bb.get(token,density)
                  case "nask" => bb.nask(token,density)
                }
}

def exec_l_choice(lst : List[(Expr,Int)]) : Int = {
        bb.l_choice(lst)
}

// exec agent
```

```
def exec(db_ag_parsed : Expr) : Boolean = {
        db_ag_parsed match {
                case DB_AST_Empty_Agent() => {true}
                case DB_AST_Primitive(db_prim,token,density) => {
                        exec_primitive(db_prim,token,density);
                         }
                case DB_AST_Agent(";",ag_i,ag_ii)
                        => { if (exec(ag_i)) { exec(ag_ii) } else { false} }
                case DB_AST_Agent("||",ag_i,ag_ii) => {
                        val t1 = thread(exec(ag_i))
                        val t2 = thread(exec(ag_ii))
                        t1.join
                        t2.join
                        true
                }
                case DB_AST_Agent("+",ag_i,ag_ii) => {
                        var lstEE = ag_first_steps(DB_AST_Agent("+",ag_i,ag_ii))
                        var lstEV = vect_ag_first_steps(lstEE)
                        var lstEI = lindex_ag_first_steps(lstEE,0)
                        var i = exec_l_choice(Random.shuffle(lstEI))
                        exec(lstEV(i))
                        true
                }
        }
   }

// generalized exec

def exec_gen(db_ag_parsed : Expr,cpt : Int) = {
   exec(db_ag_parsed)
   bb.synchronized{
        println(">> Request " + cpt + " successfully terminated")
        println()
        print("DBach> ")
   }
 }

}

/* Object MYSimInLine containing the main method of the command line simulator */

object MYSimInLine {

   def main(args: Array[String]) {

        println()
        print(" Welcome to Dense Bach version 1.\n")
        print(" Type in agents to evaluate them.\n\n")
        print("DBach> ")

        var myag_parsed : Expr = DB_AST_Empty_Agent()
        var line = readLine()
        var cpt : Int = 1
```

```
val maxHist = 5
var history = ArrayBuffer("m0", "m1", "m2", "m3", "m4")
var chist = 0
var ca : String = " "

while(line != "halt.") {
        line match {
           case "history." => { bb.synchronized{
                 println("history");
                 println()
     println("   " + "  !: " + history((maxHist+chist-1) % maxHist))
             println("   " + " !!: " + history((maxHist+chist-2) % maxHist))
                 println("   " + "!!!: " + history((maxHist+chist-3) % maxHist))
             println("   " + " !v: " + history((maxHist+chist-4) % maxHist))
             println("   " + "  v: " + history((maxHist+chist-5) % maxHist))
             println()
             print("DBach> ")
                          }
                  line = readLine()
                                }
              case "!" => { bb.synchronized{
                                println()
                                print("DBach> history mode : ")
                                println(history((chist-1) % maxHist))
                                print("DBach>> execute (y/n) : ")
                                ca = readLine()
                                if (ca == "y") {
                                     line = history((chist-1) % maxHist) + "."
                                     println()
                                } else {
                                     println()
                                     print("DBach> ")
                                }
                             }
                             if (ca != "y") { line = readLine() }
                           }
              case "!!" => { bb.synchronized{
                                println()
                                print("DBach> history mode : ")
                                println(history((chist-2) % maxHist))
                                print("DBach>> execute (y/n) : ")
                                ca = readLine()
                                if (ca == "y") {
                                     line = history((chist-2) % maxHist) + "."
                                     println()
                                } else {
                                     println()
                                     print("DBach> ")
                                }
                             }
                             if (ca != "y") { line = readLine() }
                           }
              case "!!!" => { bb.synchronized{
                                println()
                                print("DBach> history mode : ")
```

```
                println(history((chist-3) % maxHist))
                print("DBach>> execute (y/n) : ")
                ca = readLine()
                if (ca == "y") {
                    line = history((chist-3) % maxHist) + "."
                    println()
                } else {
                    println()
                    print("DBach> ")
                }
            }
            if (ca != "y") { line = readLine() }
        }
case "iv" => { bb.synchronized{
                println()
                print("DBach> history mode : ")
                println(history((chist-4) % maxHist))
                print("DBach>> execute (y/n) : ")
                ca = readLine()
                if (ca == "y") {
                    line = history((chist-4) % maxHist) + "."
                    println()
                } else {
                    println()
                    print("DBach> ")
                }
            }
            if (ca != "y") { line = readLine() }
        }
case "v" => { bb.synchronized{
                println()
                print("DBach> history mode : ")
                println(history((chist-5) % maxHist))
                print("DBach>> execute (y/n) : ")
                ca = readLine()
                if (ca == "y") {
                    line = history((chist-5) % maxHist) + "."
                    println()
                } else {
                    println()
                    print("DBach> ")
                }
            }
            if (ca != "y") { line = readLine() }
        }
case "clear." => {bb.clear_store
                  bb.synchronized{
                      println()
                      print("DBach> ")
                  }
                  line = readLine()
                  }
case "print." => {bb.print_store
                  bb.synchronized{
                      println()
```

```
                                        print("DBach> ")
                                }
                                  line = readLine()
                        }
                        case _ => {

                                while(!line.endsWith(".")) {
                                        line += readLine("    |    ")
                                }
                                line = line.replace(".","")
                                history(chist % maxHist) = line
                                chist = (chist + 1)

                                try {
                                        myag_parsed = DenseBachSimulParser.parse_agent(line)
                                        val mysimul = new DB_Exec(myag_parsed)
                                        println("DBach> >> Request " + cpt + " launched")
                        val t = mysimul.thread(mysimul.exec_gen(myag_parsed,cpt))
                                        bb.synchronized{
                                                println()
                                                print("DBach> ")
                                        }
                                        line = readLine()
                                }
                                catch {
                                        case unknown : Throwable => {
                                                bb.synchronized{
                                                        println(">> Error of parsing")
                                                        println()
                                                        print("DBach> ")
                                                }
                                                line = readLine()
                                        }
                                }
                                cpt = cpt + 1
                        }
                }
        }
    }
}
```

Appendix D

The Vectorized Dense Bach Language

D.1 The interpreter

D.1.1 The data

```
class dtExpr
case class dt(tok: String, dens: Int) extends dtExpr

class vDBachExpr
case class vdbach_ast_empty_agent() extends vDBachExpr
case class vdbach_ast_primitive(primitive: String, lDenseToken: List[dt])
                                                    extends vDBachExpr
case class vdbach_ast_agent(op: String, agenti: vDBachExpr, agentii: vDBachExpr)
                                                    extends vDBachExpr
```

D.1.2 The parser

```
import scala.util.parsing.combinator._
import scala.util.matching.Regex

class VDenseBachParsers extends RegexParsers {

  def token       : Parser[String] = ("[a-z][0-9a-zA-Z_]*").r ^^ {_.toString}

  def density     : Parser[Int] = ("[1-9][0-9]*").r ^^ {_.toInt}

  def denseToken : Parser[dt] = token~"("~density~")" ^^ {
            case vtoken ~ _ ~ vdensity ~ _ => dt(vtoken,vdensity)
                                                        }
  def vectDenseTokenList: Parser[List[dt]] = denseToken ~ rep(","
                                                    ~ vectDenseTokenList) ^^ {
                        case vdenseToken ~ List() => List(vdenseToken)
            case vdenseToken ~ List(op~lvdt) => List(vdenseToken):::lvdt
                        }
```

```
  val opChoice  : Parser[String] = "+"
  val opPara    : Parser[String] = "||"
  val opSeq     : Parser[String] = ";"

  def primitive : Parser[vDBachExpr]   = "tell("~vectDenseTokenList~")" ^^ {
        case _ ~ vvectDenseTokenList ~ _
                 => vdbach_ast_primitive("tell",vvectDenseTokenList) } |
                                  "ask("~vectDenseTokenList~")" ^^ {
        case _ ~ vvectDenseTokenList ~ _
                 => vdbach_ast_primitive("ask",vvectDenseTokenList) } |
                                  "get("~vectDenseTokenList~")" ^^ {
        case _ ~ vvectDenseTokenList ~ _
                 => vdbach_ast_primitive("get",vvectDenseTokenList) } |
                                  "nask("~vectDenseTokenList~")" ^^ {
        case _ ~ vvectDenseTokenList ~ _
                 => vdbach_ast_primitive("nask",vvectDenseTokenList) }

  def agent = compositionChoice

  def compositionChoice : Parser[vDBachExpr] =
                       compositionPara~rep(opChoice~compositionChoice) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => vdbach_ast_agent(op,agi,agii) }

  def compositionPara : Parser[vDBachExpr] =
                           compositionSeq~rep(opPara~compositionPara) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => vdbach_ast_agent(op,agi,agii) }

  def compositionSeq : Parser[vDBachExpr] =
                                  simpleAgent~rep(opSeq~compositionSeq) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => vdbach_ast_agent(op,agi,agii) }

  def simpleAgent : Parser[vDBachExpr] = primitive | parenthesizedAgent

  def parenthesizedAgent : Parser[vDBachExpr] = "("~>agent<~")"

}

object VDenseBachSimulParser extends VDenseBachParsers {

  def parse_primitive(prim: String) = parseAll(primitive,prim) match {
        case Success(result, _) => result
        case failure : NoSuccess => { scala.sys.error(failure.msg)
        }
  }

  def parse_agent(ag: String) = parseAll(agent,ag) match {
        case Success(result, _) => result
        case failure : NoSuccess => { scala.sys.error(failure.msg)
        }
  }
}
```

D.1.3 The store

```
import scala.collection.mutable.Map
import scala.swing._

class DenseBachStore {

   var theStore = Map[String,Int]()

   def tell(vectDenseTokenList:List[dt]):Boolean = {
        vectDenseTokenList match {
                case Nil => true
                case dt(tok,dens)::l => {
                        if (theStore.contains(tok)) {
                                theStore(tok) = theStore(tok) + dens
                                tell(l)
                        }
                        else
                        {       theStore = theStore ++ Map(tok -> dens)
                                tell(l)
                        }
                }
        }
   }

   def ask(vectDenseTokenList:List[dt]):Boolean = {
        vectDenseTokenList match {
                case Nil => true
                case dt(tok,dens)::l => {
                        if (theStore.contains(tok))
                                if (theStore(tok) >= dens) {
                                  ask(l)
                                }
                                else { false }
                        else false
                }
        }
   }

   def get(vectDenseTokenList:List[dt]):Boolean = {
        vectDenseTokenList match {
                case Nil => true
                case dt(tok,dens)::l => {
                        if (theStore.contains(tok))
                                if (theStore(tok) >= dens) {
                                  theStore(tok) = theStore(tok) - dens
                                  get(l)
                                }
                                else { false }
                        else false
                }
        }
   }

   def nask(vectDenseTokenList:List[dt]):Boolean = {
```

```
        vectDenseTokenList match {
                case Nil => true
                case dt(tok,dens)::l => {
                        if (theStore.contains(tok))
                                if (theStore(tok) < dens) {
                                  nask(l)
                                }
                                else { false }
                        else { nask(l) }
                }
        }
    }

    def print_store {
       print("{ ")
       for ((t,d) <- theStore)
          print ( t + "(" + theStore(t) + ") " )
       println("}")
    }

    def clear_store {
       theStore = Map[String,Int]()
    }

}

object bb extends DenseBachStore {

    def reset { clear_store }

}
```

D.1.4 The simulator

```
import scala.util.Random
import language.postfixOps

class VDenseBachSimul(var bb: DenseBachStore) {

    val vdbach_random_choice = new Random()

    def run_one(agent: vDBachExpr):(Boolean,vDBachExpr) = {

        agent match {
          case vdbach_ast_primitive(prim,vectDenseTokenList) =>
             {  if (exec_primitive(prim,vectDenseTokenList)) {
                                    (true,vdbach_ast_empty_agent()) }
                else { (false,agent) }
             }

          case vdbach_ast_agent(";",ag_i,ag_ii) =>
             {  run_one(ag_i) match
                   { case (false,_) => (false,agent)
                     case (true,vdbach_ast_empty_agent()) => (true,ag_ii)
                     case (true,ag_cont)
```

```
                => (true, vdbach_ast_agent(";", ag_cont, ag_ii))
        }
    }

case vdbach_ast_agent("||", ag_i, ag_ii) =>
    {  var branch_choice = vdbach_random_choice.nextInt(2)
       if (branch_choice == 0)
         { run_one( ag_i ) match
             { case (false, _) =>
                 { run_one( ag_ii ) match
                     { case (false, _)
                         => (false, agent)
                       case (true, vdbach_ast_empty_agent())
                         => (true, ag_i)
                       case (true, ag_cont) =>
                      (true, vdbach_ast_agent("||", ag_i, ag_cont))
                     }
                 }
               case (true, vdbach_ast_empty_agent())
                         => (true, ag_ii)
               case (true, ag_cont)
                 => (true, vdbach_ast_agent("||", ag_cont, ag_ii))
             }
         }
       else
         { run_one( ag_ii ) match
             { case (false, _) =>
                 { run_one( ag_i ) match
                     { case (false, _) => (false, agent)
                       case (true, vdbach_ast_empty_agent())
                                                 => (true, ag_ii)
                       case (true, ag_cont) =>
                      (true, vdbach_ast_agent("||", ag_cont, ag_ii))
                     }
                 }
               case (true, vdbach_ast_empty_agent())
                 => (true, ag_i)
               case (true, ag_cont)
                 => (true, vdbach_ast_agent("||", ag_i, ag_cont))
             }
         }

    }


case vdbach_ast_agent("+", ag_i, ag_ii) =>
    {  var branch_choice = vdbach_random_choice.nextInt(2)
       if (branch_choice == 0)
         { run_one( ag_i ) match
             { case (false, _) =>
                 { run_one( ag_ii ) match
                     { case (false, _) => (false, agent)
                       case (true, vdbach_ast_empty_agent())
                         => (true, vdbach_ast_empty_agent())
                       case (true, ag_cont)
```

```
                                => (true, ag_cont)
                            }
                        }
                      case (true, vdbach_ast_empty_agent())
                        => (true, vdbach_ast_empty_agent())
                      case (true, ag_cont)
                        => (true, ag_cont)
                    }
                }
              else
                { run_one( ag_ii ) match
                    { case (false, _) =>
                        { run_one( ag_i ) match
                            { case (false, _)
                                => (false, agent)
                              case (true, vdbach_ast_empty_agent())
                                => (true, vdbach_ast_empty_agent())
                              case (true, ag_cont)
                                => (true, ag_cont)
                            }
                        }
                      case (true, vdbach_ast_empty_agent())
                        => (true, vdbach_ast_empty_agent())
                      case (true, ag_cont)
                        => (true, ag_cont)
                    }
                }
            }
        }
    }
}

def vdbach_exec_all(agent: vDBachExpr): Boolean = {

    var failure = false
    var c_agent = agent
    while ( c_agent != vdbach_ast_empty_agent() && !failure ) {
       failure = run_one(c_agent) match
            { case (false, _)          => true
              case (true, new_agent)  =>
                 { c_agent = new_agent
                   false
                 }
            }
        bb.print_store
        println("\n")
    }

    if (c_agent == vdbach_ast_empty_agent()) {
        println("Success\n")
        true
    }
    else {
        println("failure\n")
        false
    }
```

```
    }

    def exec_primitive(prim:String,vectDenseTokenList:List[dt]):Boolean = {
        prim match
          { case "tell" => bb.tell(vectDenseTokenList)
            case "ask"  => bb.ask(vectDenseTokenList)
            case "get"  => bb.get(vectDenseTokenList)
            case "nask" => bb.nask(vectDenseTokenList)
          }
    }

}

object ag extends VDenseBachSimul(bb) {

  def apply(agent: String) {
     val agent_parsed = VDenseBachSimulParser.parse_agent(agent)
     ag.vdbach_exec_all(agent_parsed)
  }
  def eval(agent:String) { apply(agent) }
  def run(agent:String) { apply(agent) }

}
```

D.2 The command line simulator

```
/*  Vector Dense Bach simulator in command line                    */
/*  - compile file command : scalac codeCompilVectorDenseBach.scala */
/*  - execute object       : scala MYSimulator                     */

import scala.tools.nsc.Settings
import scala.tools.nsc.interpreter.ILoop

import scala.util.parsing.combinator._
import scala.util.matching.Regex

import scala.util.Random
import scala.collection.mutable.Map
import scala.collection.mutable.ArrayBuffer
import scala.collection.immutable.Vector
import scala.swing._

import scala.io.StdIn.readLine;

/* Data classes for Vector Dense Bach */
class dtExpr
case class dt(tok: String, dens: Int) extends dtExpr

class vDBachExpr
case class vdbach_ast_empty_agent() extends vDBachExpr
case class vdbach_ast_primitive(primitive: String,
                          lDenseToken: List[dt]) extends vDBachExpr
case class vdbach_ast_agent(op: String, agenti: vDBachExpr,
                            agentii: vDBachExpr) extends vDBachExpr
```

```
/* Vector Dense Bach parser : class VDenseBachParsers */

class VDenseBachParsers extends RegexParsers {

  def token : Parser[String] = ("[a-z][0-9a-zA-Z_]*").r ^^ {_.toString}

  def density : Parser[Int] = ("[1-9][0-9]*").r ^^ {_.toInt}

  def denseToken : Parser[dt] = token~"("~density~")" ^^ {
              case vtoken ~ _ ~ vdensity ~ _ => dt(vtoken,vdensity)
                                                                  }
  def vectDenseTokenList: Parser[List[dt]] =
                          denseToken ~ rep("," ~ vectDenseTokenList) ^^ {
                          case vdenseToken ~ List() => List(vdenseToken)
              case vdenseToken ~ List(op~lvdt) => List(vdenseToken):::lvdt
                                      }

  val opChoice   : Parser[String] = "+"
  val opPara     : Parser[String] = "||"
  val opSeq      : Parser[String] = ";"

  def primitive : Parser[vDBachExpr] = "tell("~vectDenseTokenList~")" ^^ {
        case _ ~ vvectDenseTokenList ~ _
                  => vdbach_ast_primitive("tell",vvectDenseTokenList) }  |
                                  "ask("~vectDenseTokenList~")" ^^ {
        case _ ~ vvectDenseTokenList ~ _
                  => vdbach_ast_primitive("ask",vvectDenseTokenList) }   |
                                  "get("~vectDenseTokenList~")" ^^ {
        case _ ~ vvectDenseTokenList ~ _
                  => vdbach_ast_primitive("get",vvectDenseTokenList) }   |
                                  "nask("~vectDenseTokenList~")" ^^ {
        case _ ~ vvectDenseTokenList ~ _
                     => vdbach_ast_primitive("nask",vvectDenseTokenList) }

  def agent = compositionChoice

  def compositionChoice : Parser[vDBachExpr] =
                       compositionPara~rep(opChoice~compositionChoice) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => vdbach_ast_agent(op,agi,agii) }

  def compositionPara : Parser[vDBachExpr] =
                          compositionSeq~rep(opPara~compositionPara) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => vdbach_ast_agent(op,agi,agii) }

  def compositionSeq : Parser[vDBachExpr] =
                               simpleAgent~rep(opSeq~compositionSeq) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => vdbach_ast_agent(op,agi,agii) }

  def simpleAgent : Parser[vDBachExpr] = primitive | parenthesizedAgent

  def parenthesizedAgent : Parser[vDBachExpr] = "("~>agent<~")"
```

```
}

/* Object VDenseBachSimulParser */

object VDenseBachSimulParser extends VDenseBachParsers {

  def parse_primitive(prim: String) = parseAll(primitive,prim) match {
        case Success(result, _) => result
        case failure : NoSuccess => { scala.sys.error(failure.msg)
        }
  }

  def parse_agent(ag: String) = parseAll(agent,ag) match {
        case Success(result, _) => result
        case failure : NoSuccess => { scala.sys.error(failure.msg)
        }
  }

}

/* Object bb representing the store */

object bb {

        var mapTok = Map[String,Int]()

        // synchronized methods for tell, get, ask and nask

        def tell(vectDenseTokenList:List[dt]) = bb.synchronized {
            var list : List[dt] = vectDenseTokenList
            var s : String = reg_list(vectDenseTokenList)
            while(!(list.isEmpty)) {
                if(mapTok contains (list.head.tok)) {
                        mapTok(list.head.tok) = mapTok(list.head.tok) + list.head.dens
                    } else {
                        mapTok = mapTok ++ Map(list.head.tok -> list.head.dens)
                    }
                list = list.tail
            }
            println(">> tell("+s+") successfully terminated")
            print("        >> store :")
            print_store
            println()
            print("VDBach> ")
            bb.notifyAll()
            true
        }

        def get(vectDenseTokenList:List[dt]) =  bb.synchronized {
            var list : List[dt] = vectDenseTokenList
            var s : String = reg_list(vectDenseTokenList)
            while(!ag_eval(vectDenseTokenList)) {
                println("Get waiting")
                print("VDBach> ")
```

```
            bb.wait()}
        while(!(list.isEmpty)) {
            mapTok(list.head.tok) = mapTok(list.head.tok) - list.head.dens
            list = list.tail
        }
        println(">> get("+s+") successfully terminated")
        print("        >> store :")
        print_store
        println()
        print("VDBach> ")
        bb.notifyAll()
        true
    }

    def ask(vectDenseTokenList:List[dt]) = bb.synchronized {
        var list : List[dt] = vectDenseTokenList
        var s : String = reg_list(vectDenseTokenList)
        while(!ag_eval(vectDenseTokenList)) {
                    println("Ask waiting")
                    print("VDBach> ")
                    bb.wait()}
        println(">> ask("+s+") successfully terminated")
        print("        >> store :")
        print_store
        println()
        print("VDBach> ")
        true
    }

    def nask(vectDenseTokenList:List[dt]) = bb.synchronized {
        var list : List[dt] = vectDenseTokenList
        var s : String = reg_list(vectDenseTokenList)
        while(!n_eval(vectDenseTokenList)) {
                    println("Nask  waiting")
                    print("VDBach> ")
                    bb.wait()}
        println(">> nask("+s+") successfully terminated")
        print("        >> store :")
        print_store
        println()
        print("VDBach> ")
        true
    }

    def test_tell(vectDenseTokenList:List[dt]) : Boolean = {
        var list : List[dt] = vectDenseTokenList
        var s : String = reg_list(vectDenseTokenList)
        while(!(list.isEmpty)) {
            if(mapTok contains (list.head.tok)) {
                    mapTok(list.head.tok) = mapTok(list.head.tok) + list.head.dens
                } else {
                    mapTok = mapTok ++ Map(list.head.tok -> list.head.dens)
                }
            list = list.tail
        }
```

```
    println(">> tell("+s+") successfully terminated")
    print("        >> store :")
    print_store
    println()
    print("VDBach> ")
    true
}

def test_get(vectDenseTokenList:List[dt]) : Boolean = {
    var list : List[dt] = vectDenseTokenList
    var s : String = reg_list(vectDenseTokenList)
    if(!ag_eval(vectDenseTokenList)) {
        return false} else {
    while(!(list.isEmpty)) {
        mapTok(list.head.tok) = mapTok(list.head.tok) - list.head.dens
        list = list.tail
    }
    println(">> get("+s+") successfully terminated")
    print("        >> store :")
    print_store
    println()
    print("VDBach> ")
    true }
}

def test_ask(vectDenseTokenList:List[dt]) : Boolean = {
    var list : List[dt] = vectDenseTokenList
    var s : String = reg_list(vectDenseTokenList)
    if(!ag_eval(vectDenseTokenList)) {
                return false} else {
    println(">> ask("+s+") successfully terminated")
    print("        >> store :")
    print_store
    println()
    print("VDBach> ")
    true }
}

def test_nask(vectDenseTokenList:List[dt]) : Boolean = {
    var list : List[dt] = vectDenseTokenList
    var s : String = reg_list(vectDenseTokenList)
    if(!n_eval(vectDenseTokenList)) {
                return false} else {
    println(">> nask("+s+") successfully terminated")
    print("        >> store :")
    print_store
    println()
    print("VDBach> ")
    true }
}

// Boolean function for execution of previous Dense Bach primitives

def exec_vdb_prim(prim : vDBachExpr) : Boolean = {
        prim match {
```

```
                case vdbach_ast_primitive(vdb_prim,ldt) => {
                        vdb_prim match {
                                case "tell" => {
                                        test_tell(ldt)
                                }
                                case "get" => {
                                        test_get(ldt)
                                }
                                case "ask" => {
                                        test_ask(ldt)
                                }
                                case "nask" => {
                                        test_nask(ldt)
                                }
                        }
                }
        }
}

// find an executable primitive in the vector of choices

def run_l_choice(lstEI : List[(vDBachExpr,Int)]) : (Boolean,Int) = {
        var found : Boolean = false
        var i : Int = -1
        var lst = lstEI
        while( !found && !lst.isEmpty) {
                if (exec_vdb_prim((lst.head)._1)) {
                        found = true
                        i = (lst.head)._2
                } else {
                        lst = lst.tail
                }
        }
        return (found,i)
}

// run the list of choices

def l_choice(lst : List[(vDBachExpr,Int)]) : Int = bb.synchronized {
        var r = run_l_choice(lst)
        while(! r._1 ) {
                bb.wait()
                r = run_l_choice(lst)
        }
        return r._2
}

// printing of the store content

def print_store {
        print("_{_")
        for ((t,d) <- mapTok)
                print ( t + "(" + mapTok(t) + ")_" )
        println("}")
}
```

```
// evaluation of list of tokens for get and ask

def ag_eval(ldt : List[dt]) : Boolean = {
        var l : List[dt] = ldt
        var ack : Boolean = true
        while(ack && !(l.isEmpty)) {
          if((mapTok contains (l.head.tok))
                              && (mapTok(l.head.tok) >= l.head.dens)) {
                l = l.tail
          } else {
                ack = false
          }
        }
        return ack
}

// evaluation of list of tokens for nask

def n_eval(ldt : List[dt]) : Boolean = {
        var l : List[dt] = ldt
        var ack : Boolean = true
        while(ack && !(l.isEmpty)) {
          if(!(mapTok contains (l.head.tok))
                              || (mapTok(l.head.tok) < l.head.dens)) {
                l = l.tail
          } else {
                ack = false
          }
        }
        return ack
}

// register list of dense tokens

def reg_list(ldt : List[dt]) : String = {
        var l : List[dt] = ldt
        var s : String = ""
        while(!(l.isEmpty)) {
                s = s + l.head.tok + "(" + l.head.dens + ")"
                l = l.tail
                if(!(l.isEmpty)) {
                        s = s + ","
                }
        }
        return s
}

// resetting of the store content to 0

def clear_store {
        mapTok = Map[String,Int]()
        print_store
}
```

```
}

/* class Vector DB_Exec for the execution of a Vector Dense Bach agent */

class VDB_Exec(var current_agent : vDBachExpr) {

// function for invoking thread

def thread(body: => Unit): Thread = {
     val t = new Thread {
        override def run() = body
     }
     t.start
     t
}

// function ext_continuation, for operateur op,
// add expr at the end of component el._2 of el
// el._1 contains the first step

def ext_continuation(lst : List[(vDBachExpr,vDBachExpr)],
      exp : vDBachExpr, op : String) : List[(vDBachExpr,vDBachExpr)] = {
         lst match {
               case Nil => Nil
               case el::lst_rem => (el._1,vdbach_ast_agent(op,el._2,exp))
                                          :: ext_continuation(lst_rem,exp,op)
        }
}

// function ag_first_steps transforms an agent in a list of pairs
// Expression-Expression
// The first one is the first step; the second one is its continuation

def ag_first_steps(db_ag : vDBachExpr) : List[(vDBachExpr,vDBachExpr)] = {

        db_ag match {

           // a primitive is the first step followed by the empty agent
           case vdbach_ast_primitive(vdb_prim,ldt) => {
                (vdbach_ast_primitive(vdb_prim,ldt),vdbach_ast_empty_agent())::Nil
           }

           // for choice agent, recursive call of the function for every element
           // of the choice
           case vdbach_ast_agent("+",ag_i,ag_ii) => {
                ag_first_steps(ag_i):::ag_first_steps(ag_ii)
           }

           // a sequence distinguishes both parts of Expression-Expression
           case vdbach_ast_agent(";",ag_i,ag_ii) => {
                ext_continuation(ag_first_steps(ag_i),ag_ii,";")
           }

           // for parallel agent, ag_i ith its continuity in parallel
           // with ag_ii, and vice versa
```

```
            case vdbach_ast_agent("||",ag_i,ag_ii) => {
                ext_continuation(ag_first_steps(ag_i),ag_ii,"||")
                                ::: ext_continuation(ag_first_steps(ag_ii),ag_i,"||")
            }

        }

}

// construction of Vector containing the continuations (el._2) after
// first step (el._1)

def vect_ag_first_steps(lst : List[(vDBachExpr,vDBachExpr)])
                                                    : Vector[vDBachExpr] = {
        lst match {
                case Nil => Vector.empty
                case el::lst_rem => el._2 +: vect_ag_first_steps(lst_rem)
        }
}

// Association between first step and index in the Vector

def lindex_ag_first_steps(lst : List[(vDBachExpr,vDBachExpr)], i : Int)
                                                : List[(vDBachExpr,Int)] = {
        var j = i
        lst match {
                case Nil => Nil
                case el::lst_rem => (el._1,j)::lindex_ag_first_steps(lst_rem, j + 1)
        }
}

// exec of Vector Dense Bach primitives; invokation of functions of bb

def exec_primitive(vdb_prim : String, ldt : List[dt]) = {
        vdb_prim match
                { case "tell" => bb.tell(ldt)
                  case "ask"  => bb.ask(ldt)
                  case "get"  => bb.get(ldt)
                  case "nask" => bb.nask(ldt)
                }
}

def exec_l_choice(lst : List[(vDBachExpr,Int)]) : Int = {
        bb.l_choice(lst)
}

// exec agent

def exec(vdb_ag_parsed : vDBachExpr) : Boolean = {
        vdb_ag_parsed match {
                case vdbach_ast_empty_agent() => {true}
                case vdbach_ast_primitive(vdb_prim,ldt) => {
                        exec_primitive(vdb_prim,ldt);
                         }
                case vdbach_ast_agent(";",ag_i,ag_ii) => {
```

```
                    exec(ag_i)
                    exec(ag_ii)
                    true }
                case vdbach_ast_agent("||",ag_i,ag_ii) => {
                        val t1 = thread(exec(ag_i))
                        val t2 = thread(exec(ag_ii))
                        t1.join
                        t2.join
                        true
                }
                case vdbach_ast_agent("+",ag_i,ag_ii) => {
                        var lstEE = ag_first_steps(vdbach_ast_agent("+",ag_i,ag_ii))
                        var lstEV = vect_ag_first_steps(lstEE)
                        var lstEI = lindex_ag_first_steps(lstEE,0)
                        var i = exec_l_choice(Random.shuffle(lstEI))
                        exec(lstEV(i))
                        true
                }
        }
    }

// generalized exec

def exec_gen(vdb_ag_parsed : vDBachExpr,cpt : Int) = {
    exec(vdb_ag_parsed)
    bb.synchronized{
        println(">> Request " + cpt + " successfully terminated")
        println()
        print("VDBach> ")
    }
 }

}

/* Object MYSimInLine containing the main method of the command line Simulator */

object MYSimInLine {

    def main(args: Array[String]) {

        println()
        print(" Welcome to Vector Dense Bach version 1.\n")
        print(" Type in agents to evaluate them.\n\n")
        print("VDBach> ")

        var myag_parsed : vDBachExpr = vdbach_ast_empty_agent()
        var line = readLine()
        var cpt : Int = 1

        val maxHist = 5
        var history = ArrayBuffer("m0", "m1", "m2", "m3", "m4")
        var chist = 0
        var ca : String = " "

        while(line != "halt.") {
```

```
      line match {
case "history." => { bb.synchronized{
 println("history");
 println()
 println("   " + "  !: " + history((maxHist+chist-1) % maxHist))
 println("   " + " !! : " + history((maxHist+chist-2) % maxHist))
 println("   " + "!!! : " + history((maxHist+chist-3) % maxHist))
 println("   " + " !v : " + history((maxHist+chist-4) % maxHist))
 println("   " + "  v : " + history((maxHist+chist-5) % maxHist))
 println()
 print("VDBach> ")
                }
              line = readLine()
                                }
             case "!" => { bb.synchronized{
                                println()
                                print("VDBach> history mode : ")
                                println(history((chist-1) % maxHist))
                                print("VDBach>> execute (y/n) : ")
                                ca = readLine()
                                if (ca == "y") {
                                      line = history((chist-1) % maxHist) + "."
                                      println()
                                } else {
                                      println()
                                      print("VDBach> ")
                                }
                              }
                             if (ca != "y") { line = readLine() }
                             }
             case "!!" => { bb.synchronized{
                                println()
                                print("VDBach> history mode : ")
                                println(history((chist-2) % maxHist))
                                print("VDBach>> execute (y/n) : ")
                                ca = readLine()
                                if (ca == "y") {
                                      line = history((chist-2) % maxHist) + "."
                                      println()
                                } else {
                                      println()
                                      print("VDBach> ")
                                }
                              }
                            if (ca != "y") { line = readLine() }
                             }
             case "!!!" => { bb.synchronized{
                                println()
                                print("VDBach> history mode : ")
                                println(history((chist-3) % maxHist))
                                print("VDBach>> execute (y/n) : ")
                                ca = readLine()
                                if (ca == "y") {
                                      line = history((chist-3) % maxHist) + "."
                                      println()
```

```
                } else {
                    println()
                    print("VDBach> ")
                }
              }
            if (ca != "y") { line = readLine() }
             }
case "iv" => { bb.synchronized{
                println()
                print("VDBach> history mode : ")
                println(history((chist-4) % maxHist))
                print("VDBach>> execute (y/n) : ")
                ca = readLine()
                if (ca == "y") {
                    line = history((chist-4) % maxHist) + "."
                    println()
                } else {
                    println()
                    print("VDBach> ")
                }
              }
            if (ca != "y") { line = readLine() }
             }
case "v" => { bb.synchronized{
                println()
                print("VDBach> history mode : ")
                println(history((chist-5) % maxHist))
                print("VDBach>> execute (y/n) : ")
                ca = readLine()
                if (ca == "y") {
                    line = history((chist-5) % maxHist) + "."
                    println()
                } else {
                    println()
                    print("VDBach> ")
                }
              }
              if (ca != "y") { line = readLine() }
             }
case "clear." => {bb.clear_store
                  bb.synchronized{
                      println()
                      print("VDBach> ")
                }
                line = readLine()
                }
case "print." => {bb.print_store
                  bb.synchronized{
                      println()
                      print("VDBach> ")
                }
                line = readLine()
}
case _ => {
```

```
                        while(!line.endsWith(".")) {
                                line += readLine("   |   ")
                        }
                        line = line.replace(".","")
                        history(chist % maxHist) = line
                        chist = (chist + 1)

                        try {
                                myag_parsed = VDenseBachSimulParser.parse_agent(line)
                                val mysimul = new VDB_Exec(myag_parsed)
                                println("VDBach> >> Request " + cpt + " launched")
                val t = mysimul.thread(mysimul.exec_gen(myag_parsed,cpt))
                                bb.synchronized{
                                        println()
                                        print("VDBach> ")
                                }
                                line = readLine()
                        }
                        catch {
                                case unknown : Throwable => {
                                        bb.synchronized{
                                                println(">> Error of parsing")
                                                println()
                                                print("VDBach> ")
                                        }
                                        line = readLine()
                                }
                        }
                        cpt = cpt + 1
                    }
                }
            }
        }
    }
```

Appendix E

The MRT Language

This appendix lists the full code of the MRT interpreter. This one is constitued of the abstract classes `mrExpr` and `mrAgExpr`, the `MRTParsers` class, the `MRTStore` class and finally the `MRTSimul` class.

E.1 The interpreter

E.1.1 The data

```
class mrExpr
case class mr_tp(tok: String) extends mrExpr
case class mr_tn(tok: String) extends mrExpr

class mrAgExpr
case class mrt_ast_empty_agent() extends mrAgExpr
case class mrt_ast_primitive(Pre: List[mrExpr],
                             Post: List[mrExpr]) extends mrAgExpr
case class mrt_ast_agent(op: String, agenti: mrAgExpr,
                              agentii: mrAgExpr) extends mrAgExpr
```

E.1.2 The parser

```
import scala.util.parsing.combinator._
import scala.util.matching.Regex

class MRTParsers extends RegexParsers {

  def token : Parser[String]  = ("[a-z][0-9a-zA-Z_]*").r ^^ {_.toString}
  def atoken: Parser[mrExpr]  = "+" ~ token ^^ {
                                    case _ ~ vtoken => mr_tp(vtoken)} |
                                "-" ~ token ^^ {
                                    case _ ~ vtoken => mr_tn(vtoken)}

  def atokenLis: Parser[List[mrExpr]] = atoken ~ rep("," ~ atokenList) ^^ {
                                  case vatoken ~ List() => List(vatoken)
                             case vatoken ~ List(op~lvat) => List(vatoken):::lvat
                                                  }
```

```
  def preM: Parser[List[mrExpr]] = "{" ~ "}" ^^ { case _ ~_   => List() } |
                          "{" ~ atokenList ~ "}" ^^ { case _~lvat~_ => lvat }

  def postMR : Parser[List[mrExpr]] = "{" ~ "}" ^^ { case _ ~ _  => List() } |
                          "{" ~ atokenList ~ "}" ^^ { case _~lvat~_ => lvat }

  def primitive : Parser[mrAgExpr] = "(" ~ preMR ~ "->" ~ postMR ~ ")" ^^ {
                        case _ ~ vatokenList1 ~ _ ~ vatokenList2 ~ _ =>
                                mrt_ast_primitive(vatokenList1,vatokenList2)
                                                }

  val opChoice : Parser[String] = "+"
  val opPara   : Parser[String] = "||"
  val opSeq    : Parser[String] = ";"

  def agent = compositionChoice

  def compositionChoice : Parser[mrAgExpr]
                        = compositionPara~rep(opChoice~compositionChoice) ^^ {
                      case ag ~ List() => ag
                      case agi ~ List(op~agii)  => mrt_ast_agent(op,agi,agii) }

  def compositionPara : Parser[mrAgExpr]
                              = compositionSeq~rep(opPara~compositionPara) ^^ {
                      case ag ~ List() => ag
                      case agi ~ List(op~agii)  => mrt_ast_agent(op,agi,agii) }

  def compositionSeq : Parser[mrAgExpr]
                              = simpleAgent~rep(opSeq~compositionSeq) ^^ {
                      case ag ~ List() => ag
                      case agi ~ List(op~agii)  => mrt_ast_agent(op,agi,agii) }

  def simpleAgent : Parser[mrAgExpr] = primitive | parenthesizedAgent

  def parenthesizedAgent : Parser[mrAgExpr] = "("~>agent<~")"

}


object MRTSimulParser extends MRTParsers {

   def parse_primitive(prim: String) = parseAll(primitive,prim) match {
         case Success(result, _) => result
         case failure : NoSuccess => scala.sys.error(failure.msg)
   }

   def parse_agent(ag: String) = parseAll(agent,ag) match {
         case Success(result, _) => result
         case failure : NoSuccess => scala.sys.error(failure.msg)
   }
}
```

E.1.3 The store

```
import scala.collection.mutable.Map
```

```
import scala.swing._


class MrtStore {

   var theStore = Map[String,Int]()

   // add a signed token to the adequate pre-consitions lists

   def add_to_pre_lists(atoken: mrExpr) {

      atoken match {

         case  mr_tp(x) => {
                if (thePosPre.contains(x))
                      { thePosPre(x) = thePosPre(x) + 1 }
                else
                      { thePosPre = thePosPre ++ Map(x -> 1) }
          }

         case  mr_tn(x) => {
                if (! theNegPre.contains(x))
                      { theNegPre = theNegPre ++ Map(x -> 1) }
                else
                      { theNegPre = theNegPre ++ Map(x -> 1) }
          }

         case _ => { println("error_in_precondition") }

       }
   }

   // add a signed token to the adequate post-consitions lists

   def add_to_post_lists(atoken: mrExpr) {

      atoken match {

         case  mr_tp(x) => {
                if (thePosPost.contains(x))
                      { thePosPost(x) = thePosPost(x) + 1 }
                else
                      { thePosPost = thePosPost ++ Map(x -> 1) }
          }

         case  mr_tn(x) => {
                if (theNegPost.contains(x))
                      { theNegPost(x) = theNegPost(x) + 1 }
                else
                      { theNegPost = theNegPost ++ Map(x -> 1) }
          }

         case _ => { println("error_in_postcondition") }

       }
```

```
}

// construct the pre-condition mappings, negative and positive

def pre_to_prelist(latoken: List[mrExpr]) {

      thePosPre = Map[String,Int]()
      theNegPre = Map[String,Int]()

      for (a <- latoken) { add_to_pre_lists(a) }

}

// construct the post-condition mappings, negative and positive

def pos_to_postlist(latoken: List[mrExpr]) {

      thePosPost = Map[String,Int]()
      theNegPost = Map[String,Int]()

      for (a <- latoken) { add_to_post_lists(a) }

}

// ask request on the store

def ask(token:String,number:Int): Boolean = {

   if (theStore.contains(token)) {
       if (theStore(token) >= number) { true }
       else { false }
   }
   else { false }
}

// nask request on the store

def nask(token:String,number:Int):Boolean = {

   if (theStore.contains(token))
          if (theStore(token) >= 1) { false }
          else { true }
   else true
}

// tell request on the store

def tell(token:String,number:Int):Boolean = {

   if (theStore.contains(token))
     { theStore(token) = theStore(token) + number }
   else
     { theStore = theStore ++ Map(token -> number) }
   true
}
```

```
// get request on the store

def get(token:String,number:Int):Boolean = {

   if (theStore.contains(token)) {
       if (theStore(token) >= number )
          { theStore(token) = theStore(token) - number
            true
          }
       else { println("density not enough")
          false }
   }
   else {
         println("no token present")
        false }
}

// check the feasibility of pre-conditions

def Pre(latoken: List[mrExpr]):Boolean = {

     var pre_eval = true

     pre_to_prelist(latoken)

     for ( (t,v) <- thePosPre ) {
        if ( pre_eval ) { pre_eval = ask(t,v) }
     }

     for ( (t,v) <- theNegPre ) {
        if ( pre_eval ) { pre_eval = nask(t,v) }
     }

     pre_eval

}

// check the feasibility of post-conditions

def Post(latoken: List[mrExpr]):Boolean = {

     var post_eval = true

     pos_to_postlist(latoken)

     for ( (t,v) <- thePosPost ) {
        if ( post_eval ) { post_eval = tell(t,v) }
     }

     for ( (t,v) <- theNegPost ) {
        if ( post_eval ) { post_eval = get(t,v) }
     }

     post_eval
```

```
    }

    // execution of a primitive

    def execution_primitive(mrPre: List[mrExpr],mrPost: List[mrExpr]):Boolean = {

         var thePosPre = Map[String,Int]()
         var theNegPre = Map[String,Int]()

         var thePosPost = Map[String,Int]()
         var theNegPost = Map[String,Int]()

         if(Pre(mrPre)) { Post(mrPost)
                          true }
         else { false }
    }

    // print of the state of the store

    def print_store {
       print("{ ")
       for ((t,d) <- theStore)
          print ( t + "(" + theStore(t) + ") " )
       println("}")
    }

    // make the store empty

    def clear_store:Boolean = {

       theStore = Map[String,Int]()
       true

    }

}

object bb extends MrtStore {

    def reset { clear_store }

}
```

E.1.4 The simulator

```
import scala.util.Random
import language.postfixOps

class MRTSimul(var bb: MrtStore) {

    val mrt_random_choice = new Random()

    def run_one(agent: mrAgExpr):(Boolean,mrAgExpr) = {

       agent match {
```

```
case mrt_ast_primitive(latoken1,latoken2) =>
   {  if (exec_primitive(latoken1,latoken2)) {
                              (true,mrt_ast_empty_agent()) }
      else { (false,agent) }
   }

case mrt_ast_agent(";",ag_i,ag_ii) =>
   {  run_one(ag_i) match
        { case (false,_)          => (false,agent)
          case (true,mrt_ast_empty_agent())   => (true,ag_ii)
          case (true,ag_cont)
                  => (true,mrt_ast_agent(";",ag_cont,ag_ii))
        }
   }

case mrt_ast_agent("||",ag_i,ag_ii) =>
   {  var branch_choice = mrt_random_choice.nextInt(2)
      if (branch_choice == 0)
        { run_one( ag_i ) match
            { case (false,_)          =>
                { run_one( ag_ii ) match
                    { case (false,_)    => (false,agent)
                      case (true,mrt_ast_empty_agent())
                                                  => (true,ag_i)
                      case (true,ag_cont)
                     => (true,mrt_ast_agent("||",ag_i,ag_cont))
                    }
                }
              case (true,mrt_ast_empty_agent())
                                                  => (true,ag_ii)
              case (true,ag_cont)
                  => (true,mrt_ast_agent("||",ag_cont,ag_ii))
            }
        }
      else
        { run_one( ag_ii ) match
            { case (false,_)          =>
                { run_one( ag_i ) match
                    { case (false,_)    => (false,agent)
                      case (true,mrt_ast_empty_agent())
                                                  => (true,ag_ii)
                      case (true,ag_cont)
                     => (true,mrt_ast_agent("||",ag_cont,ag_ii))
                    }
                }
              case (true,mrt_ast_empty_agent()) => (true,ag_i)
              case (true,ag_cont)
                    => (true,mrt_ast_agent("||",ag_i,ag_cont))
            }
        }

   }
```

```
      case mrt_ast_agent("+",ag_i,ag_ii) =>
         {  var branch_choice = mrt_random_choice.nextInt(2)
            if (branch_choice == 0)
              { run_one( ag_i ) match
                   { case (false,_)          =>
                       { run_one( ag_ii ) match
                           { case (false,_)    => (false,agent)
                             case (true,mrt_ast_empty_agent())
                                    => (true,mrt_ast_empty_agent())
                             case (true,ag_cont)   => (true,ag_cont)
                           }
                       }
                     case (true,mrt_ast_empty_agent())
                                    => (true,mrt_ast_empty_agent())
                     case (true,ag_cont)   => (true,ag_cont)
                   }
               }
            else
              { run_one( ag_ii ) match
                   { case (false,_) =>
                       { run_one( ag_i ) match
                           { case (false,_)    => (false,agent)
                             case (true,mrt_ast_empty_agent())
                                    => (true,mrt_ast_empty_agent())
                             case (true,ag_cont)    => (true,ag_cont)
                           }
                       }
                     case (true,mrt_ast_empty_agent())
                                    => (true,mrt_ast_empty_agent())
                     case (true,ag_cont)   => (true,ag_cont)
                   }
               }
         }
  }
}

def mrt_exec_all(agent: mrAgExpr):Boolean = {

    var failure = false
    var c_agent = agent
    while ( c_agent != mrt_ast_empty_agent() && !failure ) {
       failure = run_one(c_agent) match
            { case (false,_)          => true
              case (true,new_agent)  =>
                 { c_agent = new_agent
                   false
                 }
            }
        bb.print_store
        println("\n")
    }

    if (c_agent == mrt_ast_empty_agent()) {
        println("Success\n")
        true
```

```
        }
        else {
            println("failure\n")
            false
        }
    }

    def exec_primitive(latoken1: List[mrExpr],
                                  latoken2: List[mrExpr]):Boolean = {

         if ( bb.Pre(latoken1) ) { bb.Post(latoken2); true }
         else { false }
    }
}


object ag extends MRTSimul(bb) {

  def apply(agent: String) {

     val agent_parsed = MRTSimulParser.parse_agent(agent)
     ag.mrt_exec_all(agent_parsed)
  }

  def eval(agent:String) { apply(agent) }
  def run(agent:String) { apply(agent) }

}
```

E.2 The command line simulator

```
/*  MRT simulator in command line                          */
/*  - compile file command : scalac codeCompilMRT.scala   */
/*  - execute object       : scala MYSimulator             */

import scala.tools.nsc.Settings
import scala.tools.nsc.interpreter.ILoop

import scala.util.parsing.combinator._
import scala.util.matching.Regex

import scala.util.Random
import scala.collection.mutable.Map
import scala.collection.mutable.ArrayBuffer
import scala.collection.immutable.Vector
import scala.swing._

import scala.io.StdIn.readLine;

/* Abstract class for MRT */

class mrExpr
case class mr_tp(tok: String) extends mrExpr
case class mr_tn(tok: String) extends mrExpr
```

```
class mrAgExpr
case class mrt_ast_empty_agent() extends mrAgExpr
case class mrt_ast_primitive(mrPrePos: Map[String,Int], mrPreNeg: Map[String,Int],
          mrPostPos: Map[String,Int], mrPostNeg: Map[String,Int]) extends mrAgExpr
case class mrt_ast_agent(op: String, agenti: mrAgExpr,
                                                  agentii: mrAgExpr) extends mrAgExpr

/* MRT parser : class MRTParsers */

class MRTParsers extends RegexParsers {

  def vatokenListToMaps(l:List[mrExpr]):(Map[String,Int],Map[String,Int]) = {

        var thePos = Map[String,Int]()
        var theNeg = Map[String,Int]()

        for (a <- l) { a match {

         case  mr_tp(x) => {
                if (thePos.contains(x))
                      { thePos(x) = thePos(x) + 1 }
                else
                      { thePos = thePos ++ Map(x -> 1) }
          }

         case  mr_tn(x) => {
                if (theNeg.contains(x))
                      { theNeg(x) = theNeg(x) + 1 }
                else
                      { theNeg = theNeg ++ Map(x -> 1) }
          }

         case _ => { println("error") }

       } }
        (thePos,theNeg)
  }

  def token : Parser[String] = ("[a-z][0-9a-zA-Z_]*").r ^^ {_.toString}
  def atoken : Parser[mrExpr]= "+" ~ token ^^ {
                                  case _ ~ vtoken => mr_tp(vtoken)} |
                                    "-" ~ token ^^ {
                                  case _ ~ vtoken => mr_tn(vtoken)}

  def atokenList : Parser[List[mrExpr]] = atoken ~ rep("," ~ atokenList) ^^ {
                              case vatoken ~ List() => List(vatoken)
                         case vatoken ~ List(op~lvat) => List(vatoken):::lvat
                                                }

  def preMR : Parser[List[mrExpr]] = "{" ~ "}" ^^ { case _ ~_   => List() } |
                          "{" ~ atokenList ~ "}" ^^ { case _~lvat~_ => lvat }

  def postMR : Parser[List[mrExpr]] = "{" ~ "}" ^^ { case _ ~ _  => List() } |
                          "{" ~ atokenList ~ "}" ^^ { case _~lvat~_ => lvat }
```

```
  def primitive : Parser[mrAgExpr] = "(" ~ preMR ~ "->" ~ postMR ~ ")" ^^ {
                          case _ ~ vatokenList1 ~ _ ~ vatokenList2 ~ _ => {
                                val x = vatokenListToMaps(vatokenList1)
                                val y = vatokenListToMaps(vatokenList2)
                                mrt_ast_primitive(x._1,x._2,y._1,y._2)
                                                    }
                                               }

  val opChoice : Parser[String] = "+"
  val opPara   : Parser[String] = "||"
  val opSeq    : Parser[String] = ";"

  def agent = compositionChoice

  def compositionChoice : Parser[mrAgExpr]
                    = compositionPara~rep(opChoice~compositionChoice) ^^ {
                  case ag ~ List() => ag
                  case agi ~ List(op~agii)  => mrt_ast_agent(op,agi,agii) }

  def compositionPara : Parser[mrAgExpr]
                         = compositionSeq~rep(opPara~compositionPara) ^^ {
                  case ag ~ List() => ag
                  case agi ~ List(op~agii)  => mrt_ast_agent(op,agi,agii) }

  def compositionSeq : Parser[mrAgExpr]
                               = simpleAgent~rep(opSeq~compositionSeq) ^^ {
                  case ag ~ List() => ag
                  case agi ~ List(op~agii)  => mrt_ast_agent(op,agi,agii) }

  def simpleAgent : Parser[mrAgExpr] = primitive | parenthesizedAgent

  def parenthesizedAgent : Parser[mrAgExpr] = "("~>agent<~")"

}

/* Object MRTSimulParser */

object MRTSimulParser extends MRTParsers {

   def parse_primitive(prim: String) = parseAll(primitive,prim) match {
         case Success(result, _) => result
         case failure : NoSuccess => scala.sys.error(failure.msg)
   }

   def parse_agent(ag: String) = parseAll(agent,ag) match {
         case Success(result, _) => result
         case failure : NoSuccess => scala.sys.error(failure.msg)
   }

}

/* Object bb representing the store */

object bb {
```

```
var theStore = Map[String,Int]()

// tell request on the store ************************

def tell(token:String,number:Int):Boolean = {

 if (theStore.contains(token))
  { theStore(token) = theStore(token) + number }
 else
  { theStore = theStore ++ Map(token -> number) }
 true
}

// get request on the store *********************

def get(token:String,number:Int):Boolean = {

 if (theStore.contains(token)) {
  if (theStore(token) >= number )
     { theStore(token) = theStore(token) - number
       true
     }
  else { //println("density not enough")
     false }
 }
 else {
    //println("no token present")
   false }
}

// ask request on the store ************************

def ask(token:String,number:Int): Boolean = {

 if (theStore.contains(token)) {
  if (theStore(token) >= number) { true }
  else { false }
 }
 else { false }
}

// nask request on the store ****************************

def nask(token:String,number:Int):Boolean = {

 if (theStore.contains(token))
     if (theStore(token) >= number) { false }
     else { true }
 else true
}

// function for evaluation of feasability of pre-conditions

def eval_pre(mrPrePos : Map[String,Int],
```

```
                                mrPreNeg : Map[String,Int]) : Boolean = {
        var pre_eval : Boolean = true

        for ( (t,v) <- mrPrePos ) {
                if ( pre_eval ) { pre_eval = ask(t,v) }
        }

        for ( (t,v) <- mrPreNeg ) {
                if ( pre_eval ) { pre_eval = nask(t,v) }
        }
        return pre_eval
}

def exec_mrt_prim(prim : mrAgExpr) : Boolean = {
    prim match {
        case mrt_ast_primitive(mrPrePos,mrPreNeg,
                                                mrPostPos,mrPostNeg) => {
            var post_eval = true

            if (!(eval_pre(mrPrePos,mrPreNeg))) {false} else {

            for ( (t,v) <- mrPostPos ) {
                    if ( post_eval ) { post_eval = tell(t,v)}
            }

            for ( (t,v) <- mrPostNeg ) {
                    if ( post_eval ) { post_eval = get(t,v) }
            }
            true }
        }
    }
}

// find an executable primitive in the vector of choices

def run_l_choice(lstEI : List[(mrAgExpr,Int)]) : (Boolean,Int) = {
        var found : Boolean = false
        var i : Int = -1
        var lst = lstEI
        while( !found && !lst.isEmpty) {
                if (exec_mrt_prim((lst.head)._1)) {
                        found = true
                        i = (lst.head)._2
                } else {
                        lst = lst.tail
                }
        }
        return (found,i)
}

// run the list of choices

def l_choice(lst : List[(mrAgExpr,Int)]) : Int = bb.synchronized {
        var r = run_l_choice(lst)
        while(! r._1 ) {
```

```
                        bb.wait()
                        r = run_l_choice(lst)
                }
                return r._2
        }

        // printing of the store content

        def print_store {
                print("_{_")
                for ((t,d) <- theStore)
                        print ( t + "(" + theStore(t) + ")_" )
                println("}")
        }

        // resetting of the store content to 0

        def clear_store {
                theStore = Map[String,Int]()
                print_store
        }

}

/* class MRT_Exec for the execution of a MRT agent */

class MRT_Exec(var current_agent : mrAgExpr) {

        // function for invoking thread

        def thread(body: => Unit): Thread = {
                val t = new Thread {
                override def run() = body
                }
                t.start
                t
        }

        // function ext_continuation, for operateur op,
        // add expr at the end of component el._2 of el
        // el._1 contains the first step

        def ext_continuation(lst : List[(mrAgExpr,mrAgExpr)],
                        exp : mrAgExpr, op : String) : List[(mrAgExpr,mrAgExpr)] = {
                lst match {
                        case Nil => Nil
                        case el::lst_rem => (el._1,mrt_ast_agent(op,el._2,exp))
                                                        :: ext_continuation(lst_rem,exp,op)
                }
        }

        // function ag_first_steps transforms an agent in a list
        // of pairs Expression-Expression
        // The first one is the first step; the second one is its continuation
```

```
        def ag_first_steps(mrt_ag : mrAgExpr) : List[(mrAgExpr,mrAgExpr)] = {

        mrt_ag match {

           // a primitive is the first step followed by the empty agent
           case mrt_ast_primitive(mrPrePos,mrPreNeg,mrPostpos,mrPostNeg) => {
                (mrt_ast_primitive(mrPrePos,mrPreNeg,mrPostpos,mrPostNeg),
                                                        mrt_ast_empty_agent())::Nil
           }

           // for choice agent, recursive call of the function for
           // every element of the choice
           case mrt_ast_agent("+",ag_i,ag_ii) => {
                ag_first_steps(ag_i):::ag_first_steps(ag_ii)
           }

           // a sequence distinguishes both parts of Expression-Expression
           case mrt_ast_agent(";",ag_i,ag_ii) => {
                ext_continuation(ag_first_steps(ag_i),ag_ii,";")
           }

           // for parallel agent, ag_i with its continuity in parallel with ag_ii,
           // and vice versa
           case mrt_ast_agent("||",ag_i,ag_ii) => {
                ext_continuation(ag_first_steps(ag_i),ag_ii,"||")
                                :::ext_continuation(ag_first_steps(ag_ii),ag_i,"||")
           }

        }

}

// construction of Vector containing the continuations (el._2)
// after first step (el._1)

def vect_ag_first_steps(lst : List[(mrAgExpr,mrAgExpr)]) : Vector[mrAgExpr] = {
        lst match {
                case Nil => Vector.empty
                case el::lst_rem => el._2 +: vect_ag_first_steps(lst_rem)
        }
}

// Association between first step and index in the Vector

def lindex_ag_first_steps(lst : List[(mrAgExpr,mrAgExpr)], i : Int)
                                                    : List[(mrAgExpr,Int)] = {
        var j = i
        lst match {
                case Nil => Nil
                case el::lst_rem => (el._1,j)::lindex_ag_first_steps(lst_rem, j + 1)
        }
}

// exec of MRT primitives; invokation of functions of bb
```

```
def exec_primitive(mrPrePos: Map[String,Int], mrPreNeg: Map[String,Int],
    mrPostPos: Map[String,Int], mrPostNeg: Map[String,Int]) = bb.synchronized {
        var post_eval = true
        while(!(bb.eval_pre(mrPrePos,mrPreNeg))) {bb.wait()}
        for ( (t,v) <- mrPostPos ) {
           if ( post_eval ) { post_eval = bb.tell(t,v) }
        }
        for ( (t,v) <- mrPostNeg ) {
           if ( post_eval ) { post_eval = bb.get(t,v) }
        }
        bb.notifyAll()
        true
}

def exec_l_choice(lst : List[(mrAgExpr,Int)]) : Int = {
        bb.l_choice(lst)
}

// exec agent

def exec(mrt_ag_parsed : mrAgExpr) : Boolean = {
        mrt_ag_parsed match {
                case mrt_ast_empty_agent() => {true}
                case mrt_ast_primitive(mrPrePos,mrPreNeg,mrPostpos,mrPostNeg) => {
                        exec_primitive(mrPrePos,mrPreNeg,mrPostpos,mrPostNeg);
                         }
                case mrt_ast_agent(";",ag_i,ag_ii) => { if (exec(ag_i))
                                                { exec(ag_ii) } else { false} }
                case mrt_ast_agent("||",ag_i,ag_ii) => {
                        val t1 = thread(exec(ag_i))
                        val t2 = thread(exec(ag_ii))
                        t1.join
                        t2.join
                        true
                }
                case mrt_ast_agent("+",ag_i,ag_ii) => {
                        var lstEE = ag_first_steps(mrt_ast_agent("+",ag_i,ag_ii))
                        var lstEV = vect_ag_first_steps(lstEE)
                        var lstEI = lindex_ag_first_steps(lstEE,0)
                        var i = exec_l_choice(Random.shuffle(lstEI))
                        exec(lstEV(i))
                        true
                }
        }
   }

// generalized exec

def exec_gen(mrt_ag_parsed : mrAgExpr,cpt : Int) = {
   exec(mrt_ag_parsed)
   bb.synchronized{
        println(">> Request " + cpt + " successfully terminated")
        println()
        print("MRT> ")
   }
```

```
 }

}

/* Object MYSimInLine containing the main method of the command line simulator */

object MYSimInLine {

   def main(args: Array[String]) {

        println()
        print("␣Welcome␣to␣MRT␣version␣1.\n")
        print("␣Type␣in␣agents␣to␣evaluate␣them.\n\n")
        print("MRT>␣")

        var myag_parsed : mrAgExpr = mrt_ast_empty_agent()
        var line = readLine()
        var cpt : Int = 1

        val maxHist = 5
        var history = ArrayBuffer("m0", "m1", "m2", "m3", "m4")
        var chist = 0
        var ca : String = "␣"

        while(line != "halt.") {
                line match {
                        case "history." => { bb.synchronized{
                         println("history");
                    println()
               println("␣␣␣" + "␣␣!␣:" + history((maxHist+chist-1) % maxHist))
                    println("␣␣␣" + "␣!!␣:" + history((maxHist+chist-2) % maxHist))
                        println("␣␣␣" + "!!!␣:" + history((maxHist+chist-3) % maxHist))
                        println("␣␣␣" + "␣!v␣:" + history((maxHist+chist-4) % maxHist))
                        println("␣␣␣" + "␣␣v␣:" + history((maxHist+chist-5) % maxHist))
                            println()
                        print("MRT>␣")
               }
                    line = readLine()
                                          }
                        case "!" => { bb.synchronized{
                                        println()
                                        print("MRT>␣␣␣history␣mode␣:␣")
                                        println(history((chist-1) % maxHist))
                                        print("MRT>>␣␣␣execute␣(y/n)␣:␣")
                                        ca = readLine()
                                        if (ca == "y") {
                                             line = history((chist-1) % maxHist) + "."
                                             println()
                                        } else {
                                             println()
                                             print("MRT>␣")
                                        }
                                    }
                                    if (ca != "y") { line = readLine() }
                                }
```

```
case "!!" => { bb.synchronized{
                println()
                print("MRT> history mode : ")
                println(history((chist -2) % maxHist))
                print("MRT>> execute (y/n) : ")
                ca = readLine()
                if (ca == "y") {
                     line = history((chist -2) % maxHist) + "."
                     println()
                } else {
                     println()
                     print("MRT> ")
                }
              }
              if (ca != "y") { line = readLine() }
            }
case "!!!" => { bb.synchronized{
                println()
                print("MRT> history mode : ")
                println(history((chist -3) % maxHist))
                print("MRT>> execute (y/n) : ")
                ca = readLine()
                if (ca == "y") {
                     line = history((chist -3) % maxHist) + "."
                     println()
                } else {
                     println()
                     print("MRT> ")
                }
              }
              if (ca != "y") { line = readLine() }
            }
case "iv" => { bb.synchronized{
                println()
                print("MRT> history mode : ")
                println(history((chist -4) % maxHist))
                print("MRT>> execute (y/n) : ")
                ca = readLine()
                if (ca == "y") {
                     line = history((chist -4) % maxHist) + "."
                     println()
                } else {
                     println()
                     print("MRT> ")
                }
              }
              if (ca != "y") { line = readLine() }
            }
case "v" => { bb.synchronized{
                println()
                print("MRT> history mode : ")
                println(history((chist -5) % maxHist))
                print("MRT>> execute (y/n) : ")
                ca = readLine()
                if (ca == "y") {
```

```
                        line = history((chist-5) % maxHist) + "."
                        println()
                    } else {
                        println()
                        print("MRT> ")
                    }
                }
                if (ca != "y") { line = readLine() }
            }
    case "clear." => {bb.clear_store
                        bb.synchronized{
                            println()
                            print("MRT> ")
                        }
                        line = readLine()
                    }
    case "print." => {bb.print_store
                        bb.synchronized{
                            println()
                            print("MRT> ")
                        }
                        line = readLine()
    }
    case _ => {

            while(!line.endsWith(".")) {
                    line += readLine("   |   ")
            }
            line = line.replace(".","")
            history(chist % maxHist) = line
            chist = (chist + 1)

            try {
                    myag_parsed = MRTSimulParser.parse_agent(line)
                    val mysimul = new MRT_Exec(myag_parsed)
                    println("MRT> >> Request " + cpt + " launched")
val t = mysimul.thread(mysimul.exec_gen(myag_parsed,cpt))
                    bb.synchronized{
                            println()
                            print("MRT> ")
                    }
                    line = readLine()
            }
            catch {
                    case unknown : Throwable => {
                            bb.synchronized{
                                    println(">> Error of parsing")
                                    println()
                                    print("MRT> ")
                            }
                            line = readLine()
                    }
            }
            cpt = cpt + 1
    }
```

```
                    }
                }
            }
        }
```

Appendix F

The Simulator

F.1 The Data sructures

```
class Expr
case class DB_AST_Empty_Agent() extends Expr
case class DB_AST_Primitive(primitive: String, token: String, density: Int)
                                                        extends Expr
case class DB_Exec_AST_Primitive(primitive: String, token: String, density: Int,
                                                    path:List[Int]) extends Expr
case class DB_AST_Agent(op: String, primitive: Expr, agent: Expr) extends Expr
```

F.2 The Parser

```
import scala.util.parsing.combinator._
import scala.util.matching.Regex

class DenseBachParsers extends RegexParsers {

  def token      : Parser[String] = ("[a-z][0-9a-zA-Z_]*").r ^^ {_.toString}
  def density    : Parser[Int] = ("[1-9][0-9]*").r ^^ {_.toInt}

  val opChoice   : Parser[String] = "+"
  val opPara     : Parser[String] = "||"
  val opSeq      : Parser[String] = ";"

  def primitive : Parser[Expr]   = "tell("~token~"("~density~"))" ^^ {
    case _ ~ vtoken ~ _ ~ vdensity ~ _ => DB_AST_Primitive("tell",vtoken,vdensity) } |
                                    "ask("~token~"("~density~"))" ^^ {
    case _ ~ vtoken ~ _ ~ vdensity ~ _ => DB_AST_Primitive("ask",vtoken,vdensity) } |
                                    "get("~token~"("~density~"))" ^^ {
    case _ ~ vtoken ~ _ ~ vdensity ~ _ => DB_AST_Primitive("get",vtoken,vdensity) } |
                                    "nask("~token~"("~density~"))" ^^ {
    case _ ~ vtoken ~ _ ~ vdensity ~ _ => DB_AST_Primitive("nask",vtoken,vdensity) }

  def agent = compositionChoice

  def compositionChoice : Parser[Expr] =
```

```
                                    compositionPara~rep(opChoice~compositionChoice) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii) => DB_AST_Agent(op,agi,agii) }

  def compositionPara : Parser[Expr] = compositionSeq~rep(opPara~compositionPara) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii) => DB_AST_Agent(op,agi,agii) }

  def compositionSeq : Parser[Expr] = simpleAgent~rep(opSeq~compositionSeq) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii) => DB_AST_Agent(op,agi,agii) }

  def simpleAgent : Parser[Expr] = primitive | parenthesizedAgent

  def parenthesizedAgent : Parser[Expr] = "("~>agent<~")"
}

object DenseBachSimulParser extends DenseBachParsers {

  def parse_primitive(prim: String) = parseAll(primitive,prim) match {
        case Success(result, _) => result
        case failure : NoSuccess => { scala.sys.error(failure.msg)
        }
  }

  def parse_agent(ag: String) = parseAll(agent,ag) match {
        case Success(result, _) => result
        case failure : NoSuccess => { scala.sys.error(failure.msg)
        }
  }

}

object TParser extends DBParsers {
  def main(args: Array[String]) {
    println("input : "+ args(0))
    val res = parseAll(agent, args(0)) match {
         case Success(result, _) => result
         case failure : NoSuccess => scala.sys.error(failure.msg)
   }
    println("output : "+res)
  }
}


class PrettyPrinter {

   def translate(db_ag: Expr):String = {

      db_ag match {

         case DB_AST_Empty_Agent() => ""

         case DB_AST_Primitive(db_prim,token,density) =>
             db_prim + "(" + token + "(" + density.toString + "))"
```

```
        case DB_AST_Agent(op, ag_i, ag_ii) =>
          "[ " + translate(ag_i) + " ] " + op + " [ " + translate(ag_ii) + " ]"

    }

  }

}
```

F.3 The Store

```
import scala.collection.mutable.Map
import scala.swing._

class DBStore {
  var theStore = Map[String,Int]()

  def tell(token:String,density:Int):Boolean = synchronized {
    if (density > 0)
      {
        if (theStore.contains(token))
          { theStore(token) = theStore(token) + density }
        else
          { theStore = theStore ++ Map(token -> density) }
        true
      }
    else
      false
  }

  def test_tell(token:String,density:Int):Boolean = true

  def ask(token:String,density:Int):Boolean = synchronized {
    if ( (density > 0) && theStore.contains(token) )
          if (theStore(token) >= density) { true }
          else { false }
    else false
  }

  def test_ask(token:String,density:Int):Boolean = synchronized {
    if ( (density > 0) && theStore.contains(token) )
          if (theStore(token) >= density) { true }
          else { false }
    else false
  }

  def get(token:String,density:Int):Boolean = synchronized {
    if ( (density > 0) && theStore.contains(token) )
          if (theStore(token) >= density)
            { theStore(token) = theStore(token) - density
              true
            }
          else { false }
```

```
      else false
   }

   def test_get(token:String,density:Int):Boolean =
                                 test_ask(token:String,density:Int)

   def nask(token:String,density:Int):Boolean = synchronized {
      if ( (density > 0) && theStore.contains(token) )
            if (theStore(token) >= density) { false }
            else { true }
      else
            if (density == 0) { false }
            else { true }
   }

   def test_nask(token:String,density:Int):Boolean = synchronized {
      if ( (density > 0) && theStore.contains(token) )
            if (theStore(token) >= density) { false }
            else { true }
      else
            if (density == 0) { false }
            else { true }
   }

   def print_store {
      for ((t,d) <- theStore)
         println ( t + "(" + theStore(t) + ")" )
   }

   def clear_store:Boolean = synchronized {
      theStore = Map[String,Int]()
      true
   }
}
```

F.4 The Dense Bach Simulator

```
import scala.util.Random

class DBSimulExec(var current_agent: Expr, var bb: DBStore) {
   val db_random_choice = new Random()

   def run_unselect(db_ag: Expr): Expr = {
      db_ag match {

         case DB_AST_Empty_Agent() => DB_AST_Empty_Agent()

         case DB_AST_Primitive(db_prim,token,density)
                  => DB_AST_Primitive(db_prim,token,density)

         case DB_Exec_AST_Primitive(db_prim,token,density,pp)
                  => DB_AST_Primitive(db_prim,token,density)

         case DB_AST_Agent(op,ag_i,ag_ii)
```

```
      => DB_AST_Agent(op,run_unselect(ag_i),run_unselect(ag_ii))
    }
}


def run_selected(db_ag: Expr,path:List[Int]): Expr = {
   db_ag match {

      case DB_AST_Empty_Agent() => DB_AST_Empty_Agent()

      case DB_AST_Primitive(db_prim,token,density)
                  => DB_AST_Primitive(db_prim,token,density)

      case DB_Exec_AST_Primitive(db_prim,token,density,pp) => {
           exec_primitive(db_prim,token,density,bb)
           DB_AST_Empty_Agent() }

      case DB_AST_Agent(";",ag_i,ag_ii) => {
          val new_ag = run_selected(ag_i,path.tail)
          if ( new_ag == DB_AST_Empty_Agent() ) { ag_ii }
          else { DB_AST_Agent(";",new_ag,ag_ii) }
      }

      case DB_AST_Agent("||",ag_i,ag_ii) => {
         if ( path.head == 1 ) {
            val new_ag_i = run_selected(ag_i,path.tail)
            val new_ag_ii = run_unselect(ag_ii)
            if ( new_ag_i == DB_AST_Empty_Agent() ) { new_ag_ii }
            else { DB_AST_Agent("||",new_ag_i,new_ag_ii) } }
         else {
            val new_ag_i = run_unselect(ag_i)
            val new_ag_ii = run_selected(ag_ii,path.tail)
            if ( new_ag_ii == DB_AST_Empty_Agent() ) { new_ag_i }
            else { DB_AST_Agent("||",new_ag_i,new_ag_ii) } }
       }

      case DB_AST_Agent("+",ag_i,ag_ii) => {
         if ( path.head == 1 ) { run_selected(ag_i,path.tail) }
         else { run_selected(ag_ii,path.tail) }
       }
    }
}


def run_one(db_ag: Expr):(Boolean,Expr) = {
   db_ag match {

      case DB_AST_Primitive(db_prim,token,density) =>
         {  if (exec_primitive(db_prim,token,density,bb))
            { (true,DB_AST_Empty_Agent()) }
            else { (false,db_ag) }
         }

      case DB_AST_Agent(";",ag_i,ag_ii) =>
         {  run_one( ag_i) match
```

```
        { case (false,_)          => (false,db_ag)
          case (true,DB_AST_Empty_Agent())   => (true,ag_ii)
          case (true,ag_cont)
                  => (true,DB_AST_Agent(";",ag_cont,ag_ii))
        }
   }

case DB_AST_Agent("||",ag_i,ag_ii) =>
   {  var branch_choice = db_random_choice.nextInt(2)
      if (branch_choice == 0)
        { run_one( ag_i ) match
            { case (false,_)          =>
                { run_one( ag_ii ) match
                    { case (false,_)     => (false,db_ag)
                      case (true,DB_AST_Empty_Agent() => (true,ag_i)
                      case (true,ag_cont)
                          => (true,DB_AST_Agent("||",ag_i,ag_cont))
                    }
                }
              case (true,DB_AST_Empty_Agent())   => (true,ag_ii)
              case (true,ag_cont)
                          => (true,DB_AST_Agent("||",ag_cont,ag_ii))
            }
        }
      else
        { run_one( ag_ii ) match
            { case (false,_)          =>
                { run_one( ag_i ) match
                    { case (false,_)     => (false,db_ag)
                      case (true,DB_AST_Empty_Agent()) => (true,ag_ii)
                      case (true,ag_cont)
                          => (true,DB_AST_Agent("||",ag_cont,ag_ii))
                    }
                }
              case (true,DB_AST_Empty_Agent())   => (true,ag_i)
              case (true,ag_cont)
                           => (true,DB_AST_Agent("||",ag_i,ag_cont))
            }
        }
   }

case DB_AST_Agent("+",ag_i,ag_ii) =>
   {  var branch_choice = db_random_choice.nextInt(2)
      if (branch_choice == 0)
        { run_one( ag_i ) match
            { case (false,_) =>
                { run_one( ag_ii ) match
                    { case (false,_) => (false,db_ag)
                      case (true,DB_AST_Empty_Agent())
                                  => (true,DB_AST_Empty_Agent())
                      case (true,ag_cont)   => (true,ag_cont)
                    }
                }
              case (true,DB_AST_Empty_Agent())
                                  => (true,DB_AST_Empty_Agent())
```

```
                    case (true, ag_cont)     => (true, ag_cont)
                  }
              }
            else
              { run_one( ag_ii ) match
                  { case (false, _)          =>
                      { run_one( ag_i ) match
                          { case (false, _)     => (false, db_ag)
                            case (true, DB_AST_Empty_Agent())
                                                => (true, DB_AST_Empty_Agent())
                            case (true, ag_cont) => (true, ag_cont)
                          }
                      }
                    case (true, DB_AST_Empty_Agent())
                                                => (true, DB_AST_Empty_Agent())
                    case (true, ag_cont)    => (true, ag_cont)
                  }
              }
          }
    }
}

def exec_primitive(db_prim: String, token: String, density: Int, bb: DBStore): Boolean = {
    db_prim match
      { case "tell" => bb.tell(token, density)
        case "ask"  => bb.ask(token, density)
        case "get"  => bb.get(token, density)
        case "nask" => bb.nask(token, density)
      }
}

def test_exec_primitive(db_prim: String, token: String, density: Int, bb: DBStore): Boolean = {
    db_prim match
      { case "tell" => bb.test_tell(token, density)
        case "ask"  => bb.test_ask(token, density)
        case "get"  => bb.test_get(token, density)
        case "nask" => bb.test_nask(token, density)
      }
}

def ag_first_steps(db_ag: Expr, path: List[Int]): Expr = {
   db_ag match {

      case DB_AST_Empty_Agent() => DB_AST_Empty_Agent()

      case DB_AST_Primitive(db_prim, token, density) =>
         {  if (test_exec_primitive(db_prim, token, density, bb))
                { DB_Exec_AST_Primitive(db_prim, token, density, path) }
            else { DB_AST_Primitive(db_prim, token, density) }
         }

      case DB_AST_Agent(";", ag_i, ag_ii)
       => DB_AST_Agent(";", ag_first_steps(ag_i, path::: List(1)), ag_ii)

      case DB_AST_Agent("||", ag_i, ag_ii)
```

```
          => DB_AST_Agent("||", ag_first_steps(ag_i, path::: List(1)),
                                                ag_first_steps(ag_ii, path::: List(2)))

        case DB_AST_Agent("+", ag_i, ag_ii)
          => DB_AST_Agent("+", ag_first_steps(ag_i, path::: List(1)),
                                                ag_first_steps(ag_ii, path::: List(2)))
      }
    }
}
```

F.5 The Interavtive Blackboard

```
import scala.swing._
import scala.swing.event._

import GridBagPanel._
import java.awt.Insets

import java.awt.Color

import scala.collection.mutable.Map
import scala.collection.mutable.ArrayBuffer


object InteractiveBlackboard extends SimpleSwingApplication {

   val blue = new java.awt.Color(219, 242, 255)
   val green = new java.awt.Color(176, 255, 226)
   val red = new java.awt.Color(255, 176, 176)

   var mybb = new DBStore

   def clear_store {
      mybb.clear_store
      store_to_label(mybb.theStore)
      println("Cleared the store")
   }

   def redisplay_store {
      store_to_label(mybb.theStore)
   }


   def tell_on_store {
      val token_arg = theCurrentStore.theStoreButtons.theSTokenField.text
      val density_arg = (theCurrentStore.theStoreButtons.theSDensityField.text).toInt
      val tres = mybb.tell(token_arg,density_arg)
      if (tres) {
         store_to_label(mybb.theStore)
         println("told " + token_arg + " with density " + density_arg)
         mybb.print_store
      }
   }
```

```
   def get_from_store {
      val token_arg = theCurrentStore.theStoreButtons.theSTokenField.text
      val density_arg = (theCurrentStore.theStoreButtons.theSDensityField.text).toInt
      val gres = mybb.get(token_arg,density_arg)
      if (gres) {
         store_to_label(mybb.theStore)
         println("got " + token_arg + " with density " + density_arg)
      }
   }

   def store_to_label(theStore:Map[String,Int]) {
      def newToken(token:String,density:Int) = {
            new Label {  text = token + "[" + density.toString + "]"
                         foreground = new java.awt.Color(0, 0, 0)
                         background = blue
                         opaque = true }
      }
      theCurrentStore.bbObj.contents.clear
      for ((t,d) <- theStore)
         { theCurrentStore.bbObj.contents += newToken(t,d) }
      theCurrentStore.bbObj.revalidate()
      theCurrentStore.bbObj.repaint()
   }

   var nb_agent = 0

   def c_n_auto_agent {
      nb_agent = nb_agent + 1
      val new_auto_agent = new InteractiveAutoAgent(nb_agent,mybb)
   }

   def c_n_inter_agent {
      nb_agent = nb_agent + 1
      val new_inter_agent = new InteractiveInterAgent(nb_agent,mybb)
   }


/* ------------------------------------------------------------------------------

   Window to introduce the store

   Buttons are :

      theStoreClearButton, to clear the store
      theTellButton, to add a specified token with a specified density
      theGetButton, density to be treated

------------------------------------------------------------------------------ */


    val theCurrentStore = new GridBagPanel {
       background = blue

       val c = new Constraints
```

```
val shouldFill = true
if (shouldFill) { c.fill = Fill.Horizontal }

val theStoreTitle = new Label { text = "Current store" }
c.weightx = 0.5
c.fill = Fill.None
c.gridx = 0
c.gridy = 0
c.gridwidth = 2
c.anchor = Anchor.West
c.insets = new Insets(5,5,5,5)
layout(theStoreTitle) = c

val theStoreClearButton = new Button { text = "Clear" }
c.anchor = Anchor.East
layout(theStoreClearButton) = c

c.anchor = Anchor.West
c.gridwidth = 1
c.fill = Fill.Horizontal

val theStrutI = new Label { text = " " }
c.gridx = 0
c.gridy = 1
layout(theStrutI) = c

val labelbb = new Label("currently empty")
val bbObj = new FlowPanel {
     background = blue
     opaque = true
     contents += labelbb
     hGap = 40
     vGap = 30
     border = Swing.EmptyBorder(15,10,10,10) }
c.gridx = 1
c.gridy = 1
layout(bbObj) = c

val theStrutII = new Label { text = " " }
c.gridx = 0
c.gridy = 2
layout(theStrutI) = c

val theStoreButtons = new FlowPanel {
   background = blue
   val theTellButton = new Button { text = "Tell" }
   val theGetButton = new Button { text = "Get" }
   val theSTokenText = new Label { text = "token : " }
   val theSTokenField = new TextField { columns = 15
                                        text = "t" }
   val theSDensityText = new Label { text = "density : " }
   val theSDensityField = new TextField { columns = 10
                                          text = "1" }
   opaque = true
   contents += theTellButton
```

```
        contents += theSTokenText
        contents += theSTokenField
        contents += theSDensityText
        contents += theSDensityField
        contents += theGetButton
        hGap = 40
        vGap = 20
        border = Swing.EmptyBorder(5,10,5,10)
      }

      c.gridx = 1
      c.gridy = 2
      layout(theStoreButtons) = c

      border = Swing.EmptyBorder(15,10,15,10)
    }

/* ------------------------------------------------------------------------------

   Window to create agents
   Buttons are :
      theCreateAgentButton to create a new agent process

------------------------------------------------------------------------------ */

    val theCreateAgentButtons = new FlowPanel {

      val theCreateAutoAgentButton = new Button { text = "New Autonomous Agent" }
      val theStrutCreateButton = new Label { text = "  " }
      val theCreateInterAgentButton = new Button { text = "New Interactive Agent" }

      background = blue
      contents += theCreateAutoAgentButton
      contents += theStrutCreateButton
      contents += theCreateInterAgentButton
    }

/* ------------------------------------------------------------------------

   Main Window

------------------------------------------------------------------------ */

   val theStrutPanelI = new FlowPanel {
        background = blue
        hGap = 40
        vGap = 20
        border = Swing.EmptyBorder(5,10,5,10)
   }

   def top = new MainFrame {
      title = "The interactive blackboard"
      contents = new BoxPanel(Orientation.Vertical) {
         background = blue
         opaque = true
```

```
        contents += theCurrentStore
        contents += theStrutPanelI
        contents += theCreateAgentButtons
        border = Swing.EmptyBorder(30,30,10,10) }
    }

/* ------------------------------------------------------------------------------

    Reactions to the buttons :

      from the current store window (theCurrentStore) :
           theStoreClearButton, theTellButton, theGetButton
      from the agent window :
           theCreateAgentButton

------------------------------------------------------------------------------ */


    listenTo( theCurrentStore.theStoreClearButton,
              theCurrentStore.theStoreButtons.theTellButton,
              theCurrentStore.theStoreButtons.theGetButton,
              theCreateAgentButtons.theCreateAutoAgentButton,
              theCreateAgentButtons.theCreateInterAgentButton )
    reactions += {
      case ButtonClicked(theCurrentStore.theStoreClearButton) => clear_store
      case ButtonClicked(theCurrentStore.theStoreButtons.theTellButton)
                                                          => tell_on_store
      case ButtonClicked(theCurrentStore.theStoreButtons.theGetButton)
                                                          => get_from_store
      case ButtonClicked(theCreateAgentButtons.theCreateInterAgentButton)
                                                          => c_n_inter_agent
      case ButtonClicked(theCreateAgentButtons.theCreateAutoAgentButton)
                                                          => c_n_auto_agent }
}
```

F.6 The Interactive Agent

```
import scala.swing._
import scala.swing.event._
import GridBagPanel._
import java.awt.Insets
import java.awt.Color
import scala.collection.mutable.Map

class InteractiveInterAgent(ag_id:Int,mybb: DBStore) extends Frame {

    val blue = new java.awt.Color(196, 226, 255)
    val green = new java.awt.Color(193, 255, 193)
    val red = new java.awt.Color(255, 176, 176)

    var agent_to_be_parsed = new String
    var agent_parsed = new Expr
    var current_agent = new Expr
    var res_agent = new Expr
```

```
var previous_agent = new Expr
var mySimulParser = new DBSimulParser
var myDBsimul = new DBSimulExec(agent_parsed,mybb)

class InteractiveStepButton(prim_txt:String,path:List[Int]) extends Button {
     this.text = prim_txt
     reactions += {
       case ButtonClicked(b) => execute_step(path) }
}

class dbWidget
case class DB_Label(txt_label: String) extends dbWidget
case class DB_Button(txt_ag: String,path:List[Int]) extends dbWidget

def translate(db_ag: Expr) {
   def translate_into_widget_list(ag: Expr): List[dbWidget] = {
      ag match {

         case DB_AST_Empty_Agent() => List( DB_Label("Empty_agent") )

         case DB_AST_Primitive(db_prim,token,density) => {
           val ag_label = db_prim + "(" + token + "," + density.toString + ")"
                List( DB_Label(ag_label) ) }

         case DB_Exec_AST_Primitive(db_prim,token,density,path) => {
           val ag_label = db_prim + "(" + token + "," + density.toString + ")"
                List( DB_Button(ag_label,path) ) }

         case DB_AST_Agent(op,ag_i,ag_ii) => {
                List( DB_Label( "[_" ) ) :::
                  translate_into_widget_list(ag_i) :::
                  List( DB_Label( "_]_" + op + "_[_" ) ) :::
                  translate_into_widget_list(ag_ii) :::
                  List( DB_Label( "_]" ) ) }
     }
   }

   def group_db_labels(txt_label:String,l_db_wi:List[dbWidget]):
                                                 (String,List[dbWidget]) = {
      l_db_wi match {

         case List() => ( txt_label, l_db_wi )

         case DB_Button(txt,path) :: l_res => ( txt_label, l_db_wi )

         case DB_Label(txt) :: lres => group_db_labels(txt_label+txt,lres)
      }
  }


   def group_db_widgets(l_db_wi: List[dbWidget]) {
      l_db_wi match {

         case List() => { }
```

```
        case DB_Button(txt_ag,path) :: l_res => {
                theAgent.theCurrentAgentField.contents +=
                                                newStepButton(txt_ag,path)
                group_db_widgets(l_res) }

        case DB_Label(txt_label) :: lres => {
                val (gen_label,ll_db_wi) = group_db_labels(txt_label,lres)
                theAgent.theCurrentAgentField.contents +=
                                                    newAgTxt(gen_label)
                group_db_widgets(ll_db_wi) }
      }
   }

   def newAgTxt(txt_arg:String) = {
         new Label {  text = txt_arg
                      foreground = new java.awt.Color(0, 0, 0)
                      background = red
                      opaque = true }
   }

   def newStepButton(prim_txt: String,path:List[Int]) = {
        new InteractiveStepButton(prim_txt,path)
   }

   /*   to be actually executed in translate   */
   /*   ------------------------------------   */

   group_db_widgets(translate_into_widget_list(db_ag))

}

def parse_agent {
   agent_to_be_parsed = theAgent.theAgentField.text
   agent_parsed = mySimulParser.parse_agent(agent_to_be_parsed)
   current_agent = myDBsimul.ag_first_steps(agent_parsed,List())
   //res_agent = current_agent
   theAgent.theCurrentAgentField.contents.clear
   translate(current_agent)
   theAgent.theCurrentAgentField.revalidate()
   theAgent.theCurrentAgentField.repaint()
   println("output : "+agent_parsed)
   println("output agent labelled : "+current_agent)
}

def execute_step(path:List[Int]) {
   res_agent = myDBsimul.run_selected(current_agent,path)
   current_agent = myDBsimul.ag_first_steps(res_agent,List())
   theAgent.theCurrentAgentField.contents.clear
   translate(current_agent)
   theAgent.theCurrentAgentField.revalidate()
   theAgent.theCurrentAgentField.repaint()
   println("new agent : "+res_agent)
   InteractiveBlackboard.redisplay_store

}
```

```
   def refresh_agent {
      current_agent = myDBsimul.ag_first_steps(res_agent,List())
      theAgent.theCurrentAgentField.contents.clear
      translate(current_agent)
      theAgent.theCurrentAgentField.revalidate()
      theAgent.theCurrentAgentField.repaint()
   }

/* ------------------------------------------------------------------------------

   Window to introduce the agent to be processed
   Buttons are :
      theSubmitAgentButton, to submit the agent ie to parse it into internal format
      theStepAgentButton, to run step by step the agent

------------------------------------------------------------------------------ */

    val theAgent = new GridBagPanel {
       background = red

       /* Agent to be processed */
       /* --------------------- */

       val c = new Constraints
       val shouldFill = true
       if (shouldFill) { c.fill = Fill.Horizontal }

       val theAgentTitle = new Label { text = "Agent to be processed" }
       c.weightx = 0.5
       c.fill = Fill.None
       c.gridx = 0
       c.gridy = 0
       c.gridwidth = 3
       c.anchor = Anchor.West
       c.insets = new Insets(5,5,5,5)
       layout(theAgentTitle) = c

       c.gridwidth = 1
       c.fill = Fill.Horizontal

       val theStrut_i = new Label { text = " " }
       c.gridx = 0
       c.gridy = 1
       layout(theStrut_i) = c

       val theAgentField = new TextArea(10, 60)
       theAgentField.text = "Enter the agent"
       val theScrollableAgentField = new ScrollPane(theAgentField)
       c.gridx = 1
       c.gridy = 1
       layout(theScrollableAgentField) = c

       val theSubmitAgentButton = new Button {
          text = "Submit" }
```

```
        c.gridx = 2
        c.gridy = 1
        layout(theSubmitAgentButton) = c

        /* Current value of the agent */
        /* ------------------------- */

        val theCurrentAgentTitle = new Label { text = "Current agent" }
        c.weightx = 0.5
        c.fill = Fill.None
        c.gridx = 0
        c.gridy = 2
        c.gridwidth = 3
        c.anchor = Anchor.West
        c.insets = new Insets(5,5,5,5)
        layout(theCurrentAgentTitle) = c

        c.gridwidth = 1
        c.fill = Fill.Horizontal

        val theStrut_ii = new Label { text = " " }
        c.gridx = 0
        c.gridy = 3
        layout(theStrut_ii) = c

        val labelbb = new Label("currently empty")
        val theCurrentAgentField = new FlowPanel {
             background = red
             opaque = true
             contents += labelbb
             hGap = 40
             vGap = 30
             border = Swing.EmptyBorder(15,10,10,10) }
        c.gridx = 1
        c.gridy = 3

        val theSCurrentAgentField = new ScrollPane(theCurrentAgentField)

        layout(theSCurrentAgentField) = c

        val theRefreshAgentButton = new Button {
           text = "Refresh" }
        c.gridx = 2
        c.gridy = 3
        layout(theRefreshAgentButton) = c
    }

/* ------------------------------------------------------------------------
   Reactions to the buttons :
     from the main agent window :
          theSubmitAgentButton, theRunAgentButton, theStepAgentButton
     from the history window :
          thePAgentButton, theNAgentButton
------------------------------------------------------------------------ */
```

```
      listenTo(theAgent.theSubmitAgentButton,
               theAgent.theRefreshAgentButton)
      reactions += {
          case ButtonClicked(theAgent.theSubmitAgentButton) => parse_agent
          case ButtonClicked(theAgent.theRefreshAgentButton) => refresh_agent
  }

/* ------------------------------------------------------------------

   Main elements

------------------------------------------------------------------ */

   val theStrutPanelI = new FlowPanel {
           background = red
           hGap = 40
           vGap = 20
           border = Swing.EmptyBorder(5,10,5,10)
       }

   val theStrutPanelII = new FlowPanel {
           background = red
           hGap = 40
           vGap = 20
           border = Swing.EmptyBorder(5,10,5,10)
       }

   this.title = "Agent number " + ag_id.toString
   this.visible = true
   this.contents = new BoxPanel(Orientation.Vertical) {
         background = red
         opaque = true
         contents += theAgent
         border = Swing.EmptyBorder(30,30,10,10) }
   this.pack()
}
```

F.7 The Autonomous Agent

```
import scala.swing._
import scala.swing.event._
import GridBagPanel._
import java.awt.Insets
import java.awt.Color
import scala.collection.mutable.Map

class InteractiveAutoAgent(ag_id:Int,mybb: DBStore) extends Frame {

   val blue = new java.awt.Color(196, 226, 255)
   val green = new java.awt.Color(193, 255, 193)
   val red = new java.awt.Color(255, 176, 176)

   var agent_to_be_parsed = new String
   var agent_parsed = new Expr
```

```
  var current_agent = new Expr
  var previous_agent = new Expr
  var mySimulParser = new DBSimulParser
  var myDBsimul = new DBSimulExec(agent_parsed,mybb)
  var myTranslator = new PrettyPrinter


  def parse_agent {
     agent_to_be_parsed = theAgent.theAgentField.text
     agent_parsed = mySimulParser.parse_agent(agent_to_be_parsed)
     current_agent = agent_parsed
     theAgent.theCurrentAgentField.text =
                              myTranslator.translate(current_agent)
     println("output : "+agent_parsed)
  }

  def execute_step:Boolean = {
      var exec_result = true
      previous_agent = current_agent
      if (current_agent != DB_AST_Empty_Agent()) {
         exec_result = myDBsimul.run_one(current_agent) match
              { case (false,_)          => false
                case (true,new_agent)  =>
                   { current_agent = new_agent
                     true
                   }
              }
         theAgent.theCurrentAgentField.text = myTranslator.translate(current_agent)
         InteractiveBlackboard.redisplay_store
      }
      exec_result
  }

  def execute_all = {
      var failure = false
      while ( current_agent != DB_AST_Empty_Agent() && !failure ) {
         println("enter execution step")
         failure = !execute_step
      }
  }

/* ------------------------------------------------------------------------

   Window to introduce the agent to be processed

   Buttons are :

      theSubmitAgentButton, to submit the agent ie to parse it into internal format
      theRunAgentButton, to run the agent from begin to end
      theStepAgentButton, to run step by step the agent

------------------------------------------------------------------------ */

   val theAgent = new GridBagPanel {
```

```
background = green

/* Agent to be processed */
/* ---------------------- */

val c = new Constraints
val shouldFill = true
if (shouldFill) { c.fill = Fill.Horizontal }

val theAgentTitle = new Label { text = "Agent to be processed" }
c.weightx = 0.5
c.fill = Fill.None
c.gridx = 0
c.gridy = 0
c.gridwidth = 3
c.anchor = Anchor.West
c.insets = new Insets(5,5,5,5)
layout(theAgentTitle) = c

c.gridwidth = 1
c.fill = Fill.Horizontal

val theStrut_i = new Label { text = " " }
c.gridx = 0
c.gridy = 1
layout(theStrut_i) = c

val theAgentField = new TextArea(10, 60)
theAgentField.text = "Enter the agent"
val theScrollableAgentField = new ScrollPane(theAgentField)
c.gridx = 1
c.gridy = 1
layout(theScrollableAgentField) = c

val theSubmitAgentButton = new Button {
   text = "Submit" }
c.gridx = 2
c.gridy = 1
layout(theSubmitAgentButton) = c

/* Current value of the agent */
/* -------------------------- */

val theCurrentAgentTitle = new Label { text = "Current agent" }
c.weightx = 0.5
c.fill = Fill.None
c.gridx = 0
c.gridy = 2
c.gridwidth = 3
c.anchor = Anchor.West
c.insets = new Insets(5,5,5,5)
layout(theCurrentAgentTitle) = c

c.gridwidth = 1
c.fill = Fill.Horizontal
```

```
      val theStrut_ii = new Label { text = "_" }
      c.gridx = 0
      c.gridy = 3
      layout(theStrut_ii) = c

      val theCurrentAgentField = new Label  {
         text = "Here will be displayed the current agent" }
      c.gridx = 1
      c.gridy = 3
      layout(theCurrentAgentField) = c

      /* The running buttons */
      /* ------------------- */

      val theRunningButtons = new FlowPanel {
         background = green
         val theRunAgentButton = new Button { text = "Run" }
         val theStepAgentButton = new Button { text = "Next" }
         opaque = true
         contents += theRunAgentButton
         contents += theStepAgentButton
         hGap = 40
         vGap = 20
         border = Swing.EmptyBorder(5,10,5,10)
      }

      c.gridx = 0
      c.gridy = 4
      c.gridwidth = 3
      layout(theRunningButtons) = c

      border = Swing.EmptyBorder(15,10,15,10)

    }

/* ------------------------------------------------------------------------

   Reactions to the buttons :

     from the main agent window :
          theSubmitAgentButton, theRunAgentButton, theStepAgentButton
     from the history window :
          thePAgentButton, theNAgentButton

------------------------------------------------------------------------ */

   listenTo(theAgent.theSubmitAgentButton,
            theAgent.theRunningButtons.theStepAgentButton,
            theAgent.theRunningButtons.theRunAgentButton)
   reactions += {
     case ButtonClicked(theAgent.theSubmitAgentButton) => parse_agent
     case ButtonClicked(theAgent.theRunningButtons.theStepAgentButton) => execute_step
     case ButtonClicked(theAgent.theRunningButtons.theRunAgentButton) => execute_all
   }
```

```
/* ------------------------------------------------------------------

   Main elements

------------------------------------------------------------------ */

   val theStrutPanelI = new FlowPanel {
          background = green
          hGap = 40
          vGap = 20
          border = Swing.EmptyBorder(5,10,5,10)
   }

   val theStrutPanelII = new FlowPanel {
          background = green
          hGap = 40
          vGap = 20
          border = Swing.EmptyBorder(5,10,5,10)
   }

   this.title = "Agent number " + ag_id.toString
   this.visible = true
   this.contents = new BoxPanel(Orientation.Vertical) {
         background = green
         opaque = true
         contents += theAgent
         border = Swing.EmptyBorder(30,30,10,10) }
   this.pack()
}
```

Appendix G

From Dense Bach to Petri Net

This appendix lists the full code of the sub-procedures used by the procedure *convert2PetriNet* of the class *PetriNetEquivalent*. This procedure converts the result of the parsing of a Dense Bach agent into a structure of Petri Net.

```
class PetriNetEquivalent {

 def convert2petriNet(agent : Expr, prefid : List[String],
    posX : Int, posY : Int) : (PetriNet,Int) = { // Int = number of tokens

 var pn = PetriNet(Nil,Nil,Nil,Nil,Nil,Nil)

 /* internal procedures for obtaining informations or for
 modifications of lists */

 def removeInPre(lpre:List[pre_condition],
        idInPlace:List[String]) :
             List[pre_condition] = { // remove pre associated to place In
  lpre match {
   case Nil => Nil
   case pre_condition(idp,idt,pds,cargr)::l => {
    if (idp == idInPlace) {
     removeInPre(l,idInPlace)
    } else {
     pre_condition(idp,idt,pds,cargr)::removeInPre(l,idInPlace)
    }
   }
  }
 }

 def removeInPlace(lp:List[place]) :
  (List[String],List[place]) = { // remove IN place in a list of places;
                  // return id of place IN  and the resulting list of places
  lp match {
   case Nil => (Nil,Nil)
   case place(cx,cy,radius,true,exit,regular,name,nbrTokens,
                nbrReq,idplace,caraGraphi)::l => (idplace,l)
```

```
  case place(cx,cy,radius,false,exit,regular,name,nbrTokens,
              nbrReq,idplace,caraGraphi)::l =>
          (removeInPlace(l)._1,place(cx,cy,radius,false,exit,
   regular,name,nbrTokens,nbrReq, idplace,caraGraphi)::removeInPlace(l)._2)
 }
}

def removeEmptyPlace(lp:List[place]) : List[place] = { // remove place with no id
 lp match {
  case Nil => Nil
  case place(cx,cy,radius,entry,exit,regular,name,nbrTokens,
                                        nbrReq,idplace,caraGraphi)::l => {
   if(idplace.isEmpty) {
    removeEmptyPlace(l)
   } else {
    place(cx,cy,radius,entry,exit,regular,name,nbrTokens,
                       nbrReq,idplace,caraGraphi)::removeEmptyPlace(l)
   }
  }
 }
}

def replacePlacePre(idPlace2BeRemoved : List[String], newIdPlace : List[String],
          pc : Int, lpre : List[pre_condition]) : List[pre_condition] = {
               // replace idPlace2BeRemoved by newIdPlace in a list of pre
 lpre match {
  case Nil => Nil
  case pre_condition(idplace, idtrans, weight, caraGraphi)::l => {
   if (idplace == idPlace2BeRemoved) { pre_condition(newIdPlace,idtrans,
      pc, caraGraphi)::replacePlacePre(idPlace2BeRemoved,newIdPlace,pc,l) }
   else { pre_condition(idplace,idtrans, weight, caraGraphi)
                     ::replacePlacePre(idPlace2BeRemoved,newIdPlace,pc,l) }
   }
  }
 }

def removeInAndReplace(id:List[String],pc:Int,
                       pn:PetriNet) : PetriNet = { // remove IN of pn with
                                                   // incidence on pre
 val (idp,lp) = removeInPlace(pn.SetOfPlaces) // idp removed from pn
 val lpre = replacePlacePre(idp,id,pc,pn.SetOfPre) // replace idp (removed) by
                                                   // id in list of pre
 PetriNet(lp,pn.SetOfTrans,lpre,pn.SetOfPost,pn.SetOfPreSpaces,pn.SetOfPostSpaces)
}

def findInPlace(lp:List[place]) : List[String] = { // find a place IN
                                                   // in the list of places
 lp match {
  case Nil => Nil
  case place(cx,cy,radius,true,exit,regular,name,nbrTokens,
                               nbrReq,idplace,caraGraphi)::l => idplace
  case place(cx,cy,radius,false,exit,regular,name,nbrTokens,
                               nbrReq,idplace,caraGraphi)::l => findInPlace(l)
 }
}
```

```
 def findOutPlace(lp:List[place]) : List[String] = { // find place OUT in
                                                     // the list of places
  lp match {
   case Nil => Nil
   case place(cx,cy,radius,entry,true,regular,name,nbrTokens,
                                  nbrReq,idplace,caraGraphi)::l => idplace
   case place(cx,cy,radius,entry,false,regular,name,nbrTokens,
                                  nbrReq,idplace,caraGraphi)::l => findOutPlace(l)
  }
 }

 def unionPetriNet(pn1:PetriNet,pn2:PetriNet) : PetriNet = {// union of two petrinet
  PetriNet(pn1.SetOfPlaces:::pn2.SetOfPlaces,
        pn1.SetOfTrans:::pn2.SetOfTrans,
        pn1.SetOfPre:::pn2.SetOfPre,
        pn1.SetOfPost:::pn2.SetOfPost,
        pn1.SetOfPreSpaces:::pn2.SetOfPreSpaces,
        pn1.SetOfPostSpaces:::pn2.SetOfPostSpaces)
 }

 def transfOutToRegPlace(lp:List[place],nReq:Int) : List[place] = { // transform an
                                                  // exit place to a regular place
  lp match {
   case Nil => Nil
   case place(cx,cy,radius,false,true,false,name,nbrTokens,
                                     nbrReq,idplace,caraGraphi)::l =>
     place(cx,cy,radius,false,false,true,name,nbrTokens,nReq,idplace,caraGraphi)::l
   case place(cx,cy,radius,entry,false,regular,name,nbrTokens,
                                  nbrReq,idplace,caraGraphi)::l =>
     place(cx,cy,radius,entry,false,regular,name,nbrTokens,
                           nbrReq,idplace,caraGraphi)::transfOutToRegPlace(l,nReq)
   }
 }

def findWeightInPlace(idplacet : List[String], lplace : List[place]) : Int = {
      // find the weight in a place idplacet (in a list of places)
  lplace match {
   case Nil => 0
   case place(cx,cy,radius,entry,exit,regular,name,nbrTokens,
                                  nbrReq,idplace,caraGraphi)::l => {
    if(idplace == idplacet) {
                nbrTokens
    } else {
                findWeightInPlace(idplacet,l)
    }
   }
  }
 }

 def replaceAddTransInPost(newTrans1 : List[String],
                      newTrans2 : List[String], trans2beReplaced : List[String],
  lpost : List[post_condition]) : List[post_condition] = {
   // remove trans2beReplaced by newTrans1 in a post AND add a new post with newTrans2
  lpost match {
```

```
    case Nil => Nil
    case post_condition(idtrans, idplace, weight, caraGraphi)::l => {
     if(idtrans == trans2beReplaced) {
      post_condition(newTrans1, idplace, weight, caraGraphi)
                                                    ::post_condition(newTrans2,
       idplace,weight, caraGraphi)
                  ::replaceAddTransInPost(newTrans1,newTrans2,trans2beReplaced,l)
     } else {
      post_condition(idtrans, idplace, weight, caraGraphi)
                  ::replaceAddTransInPost(newTrans1,newTrans2,trans2beReplaced,l)
     }
    }
   }
  }

  def collectIdPlaces(lplace:List[place]) : List[List[String]] = { // collect all
                                          //the idplaces in a list of places
   lplace match { // create a list with all the idplace present in a list of places
    case Nil => Nil
    case place(cx,cy,radius,entry,exit,regular,name,nbrTokens,
                                        nbrReq,idplace, caraGraphi)::l => {
          idplace::collectIdPlaces(l)
    }
   }
  }

  def collectIdTransPre(lpre:List[pre_condition],
                        inPlace : List[String]) : List[List[String]] = {
                // collect in a list of pre all the idtrans related
                // to a specific inPlace
   lpre match {
    case Nil => Nil
    case pre_condition(idplace, idtrans, weight, caraGraphi)::l => {
     if(idplace == inPlace) {
            idtrans::collectIdTransPre(l,inPlace)
     } else {
            collectIdTransPre(l,inPlace)
     }
    }
   }
  }

  def giveAll(idPlaceAux : List[String], lIdTrans : List[List[String]],
        weight : Int, caraGraphi : String) : List[pre_condition] = {
   lIdTrans match { // generate the list of all pre of idTrans, from the list
                // lIdTrans, for the place idPlaceAux
    case Nil => Nil
    case idTrans::l => pre_condition(idPlaceAux,idTrans,weight,
                caraGraphi)::giveAll(idPlaceAux,l,weight,caraGraphi)
   }
  }

 def newPostAux(idPlaceAux : List[String], lIdTrans : List[List[String]],
                  weight : Int) : List[post_condition] = {
   lIdTrans match { // generate new post for the auxiliary places idPlaceAux
```

```
   case Nil => Nil
   case idTrans::l => post_condition(idTrans,idPlaceAux,weight,
                  "black")::newPostAux(idPlaceAux,l,weight)
 }
}

def valueMax(a: Int, b : Int): Int = { // return max value beteen a and b
 if (a > b) {
       a
 }
 else {
       b
 }
}

def valueSum(a: Int, b: Int): Int = { // return summation of a and b
               a + b
}

/* end of internal methods */
```

Appendix H

Svg Picture of Petri Net

H.1 Subprocedures for the conversion of Petri Net to svg

This appendix lists the full code of the sub-procedures used by the procedure *convertPn2Svg* of the class *pn2xml*. This procedure converts the Petri Net structure into an svg picture, via an xml file.

```
class pn2xml(var nomFile: String) {

 var xmlFile = new PrintWriter(new File(nomFile))

 def openSvgFile() { // as no writing in file after closing,
        // open a new xml file for every step of execution of Petri Net
  xmlFile = new PrintWriter(new File(nomFile))
 }

 def close_file() { // close xml file
  xmlFile.close()
 }

 def beginSVG() { // header of the xml file with reference
                  // to text-anchor.css + definition of marker arrow

  xmlFile.write("<?xml version=\"1.0\" encoding=\"utf-8\"?>\n")
  xmlFile.write("<?xml-stylesheet type=\"text/css\"
                            href=\"text-anchor.css\" charset=\"utf-8\"?>\n")
  xmlFile.write("<!DOCTYPE svg PUBLIC \"-//W3C//DTD SVG 20010904//EN\"
                \"http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd\">\n")
  xmlFile.write("<svg width=\"1600px\" height=\"1600px\" xml:lang=\"fr\"
                xmlns=\"http://www.w3.org/2000/svg\"
                xmlns:xlink=\"http://www.w3.org/1999/xlink\">\n")
  xmlFile.write("<title>XML description of Petri Net for SVG</title>\n")

  xmlFile.write("<defs>\n")
  xmlFile.write("<marker id=\"arrow\" markerWidth=\"10\" markerHeight=\"10\"
                refX=\"0\" refY=\"4\" orient=\"auto\" markerUnits=
                \"strokeWidth\">\n")
  xmlFile.write("<path d=\"M 0,0 L 4,3.5 L 0,3.5 L 0,4.5
```

```
                                        L 4,4.5 L 0,8 L 10,4 z\" fill=\"black\" />\n")
    xmlFile.write("</marker>\n")
    xmlFile.write("</defs>\n")
  }

  def endSVG() { // end marker of the xml file
    xmlFile.write("</svg>")
  }

  /* begin of methods to draw the different components of a Petri Net */

  def drawTransition(posx : Int, posy: Int, width : Int, height : Int) {
                      // draw by default transition in black
    xmlFile.write(f"""<rect x=\"$posx%2d\" y=\"$posy%2d\" width=\"$width%2d\"
    height=\"$height%2d\" style=\"fill:none;stroke:black;stroke-width:3px;\"/>\n""")
    }

  def drawTransitionColour(posx : Int, posy: Int, width : Int, height : Int) {
                      // draw possible firable transition in red
    xmlFile.write(f"""<rect x=\"$posx%2d\" y=\"$posy%2d\" width=\"$width%2d\"
    height=\"$height%2d\" style=\"fill:none;stroke:red;stroke-width:3px;\"/>\n""")
  }

  def drawPlaceIN(posx : Int, posy: Int, radius : Int) { // draw entry place In in blue
    xmlFile.write(f"""<circle cx=\"$posx%2d\" cy=\"$posy%2d\" r=\"$radius%2d\"
        style=\"fill:white;stroke:blue;stroke-width:3px;\" />\n""")
  }

  def drawPlaceOUT(posx : Int, posy: Int, radius : Int) { // draw exit place Out in blue
    xmlFile.write(f"""<circle cx=\"$posx%2d\" cy=\"$posy%2d\" r=\"$radius%2d\"
                  style=\"fill:white;stroke:blue;stroke-width:3px;\" />\n""")
  }

  def drawPlace(posx : Int, posy: Int, radius : Int, color : String) {
                                  // draw by default a place in black
    xmlFile.write(f"""<circle cx=\"$posx%2d\" cy=\"$posy%2d\" r=\"$radius%2d\"
                  style=\"fill:white;stroke:$color%s;stroke-width:3px;\" />\n""")
  }

  def drawP2T(posix : Int, posiy : Int, posfx : Int, posfy : Int) {
                                  // draw straight line arrow from place to transition
    xmlFile.write(f"""<line x1=\"$posix%2d\" y1=\"$posiy%2d\" x2=\"$posfx%2d\" y2=
    \"$posfy%2d\" stroke=\"black\" stroke-width=\"3\" marker-end=\"url(#arrow)\"/>\n""")
  }

  def drawT2P(posix : Int, posiy : Int, posfx : Int, posfy : Int) {
                                  // draw straight line arrow from transition to place
    xmlFile.write(f"""<line x1=\"$posix%2d\" y1=\"$posiy%2d\" x2=\"$posfx%2d\" y2=
    \"$posfy%2d\" stroke=\"black\" stroke-width=\"3\" marker-end=\"url(#arrow)\"/>\n""")
  }

  def drawT2Pcourbe(posix : Int, posiy : Int, poinfx : Int, poinfy : Int, posfx : Int,
                                                    posfy : Int, color : String) {
                // draw curved line arrow from transition to place
    xmlFile.write(f"""<path d=\"M $posix%2d,$posiy%2d Q $poinfx%2d,$poinfy%2d $posfx%2d,
```

```
    $posfy%2d" style = "stroke:$color%s;stroke-width:3px;fill:none;
                                                    marker-end:url(#arrow)" />\n""")
  }

  def drawP2Tcourbe(posix : Int, posiy : Int, poinfx : Int, poinfy : Int, posfx : Int,
                                                        posfy : Int, color : String) {
                // draw curved line arrow from place to transition
  xmlFile.write(f"""<path d="M $posix%2d,$posiy%2d Q $poinfx%2d,$poinfy%2d $posfx%2d,
    $posfy%2d" style = "stroke:$color%s;stroke-width:3px;fill:none;
                                                    marker-end:url(#arrow)" />\n""")
  }

  def drawText(postx : Int, posty : Int, width : Int, height : Int, name : String,
                                                      token : String, density : Int) {
// for transition representing a primitive, draw name of primitive, token name and density
    val posX = postx + 7*width/12
    val posY = posty - 6
    xmlFile.write(f"""<text id=\"$name%s\" x=\"$posX%2d\" y=\"$posY%2d\">
                                            $name%s($token%s($density%d)) </text>\n""")
  }

  def drawTransName(postx : Int, posty : Int, width : Int, height : Int, name : String) {
                // for transition not related to a primitive, draw its name only
    val posX = postx + 7*width/12
    val posY = posty - 6
    xmlFile.write(f"""<text id=\"$name%s\" x=\"$posX%2d\" y=\"$posY%2d\"> $name%s </text>\n""")
  }

  def drawTextColour(postx : Int, posty : Int, width : Int, height : Int,
                                    name : String, key : String) {
                // draw text in colour
    val posX = postx + 7*width/12
    val posY = posty - 3
    xmlFile.write(f"""<text id=\"$name%s\" x=\"$posX%2d\" y=\"$posY%2d\">
                                                $key%s </text>\n""")
  }

  def drawTokenName(posX : Int, posY : Int, token : String) {
      // draw token name in front of its place and anti-place in token space
    xmlFile.write(f"""<text id=\"$token%s\" x=\"$posX%2d\" y=\"$posY%2d\">
                                                $token%s </text>\n""")
  }

  def drawNbrTok(posX : Int, posY : Int, density : Int) {
                // draw number of token in a place
    xmlFile.write(f"""<text id=\"middle\" x=\"$posX%2d\" y=\"$posY%2d\">
                                                $density%d </text>\n""")
  }

/* end of drawing methods */
```

H.2 Conversion of Petri Net to svg

```
def convertPn2Svg(pn : PetriNet, mp : Map[Int, List[String]]) {
                openSvgFile()
                beginSVG()

// drawing of the arrows of the pre_conditions - from place to transition

     pn.SetOfPre.foreach( elm => { // for every pre_conditions of final Petri Net
     (elm.idplace).head match { // drawing determined by type of starting place
                                 case "Auxl+" => {
drawP2Tcourbe(findCoordXOfPlace(elm.idplace,pn.SetOfPlaces)+radius,
findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)-radius/6,
findCoordXOfTrans(elm.idtrans,pn.SetOfTrans) + findWidthOfTrans(elm.idtrans,pn.SetOfTrans)/2,
findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)+(findCoordYOfTrans(elm.idtrans,pn.SetOfTrans)-
findCoordYOfPlace(elm.idplace,pn.SetOfPlaces))/3,
findCoordXOfTrans(elm.idtrans,pn.SetOfTrans) + findWidthOfTrans(elm.idtrans,pn.SetOfTrans)/2,
findCoordYOfTrans(elm.idtrans,pn.SetOfTrans)-radius,"green")
                                 }
                                 case "Auxl" => {
drawP2Tcourbe(findCoordXOfPlace(elm.idplace,pn.SetOfPlaces)+radius,
findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)-radius/6,
findCoordXOfTrans(elm.idtrans,pn.SetOfTrans) + findWidthOfTrans(elm.idtrans,pn.SetOfTrans)/2,
findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)+(findCoordYOfTrans(elm.idtrans,pn.SetOfTrans)-
findCoordYOfPlace(elm.idplace,pn.SetOfPlaces))/3,
findCoordXOfTrans(elm.idtrans,pn.SetOfTrans)+ findWidthOfTrans(elm.idtrans,pn.SetOfTrans)/2,
findCoordYOfTrans(elm.idtrans,pn.SetOfTrans)-radius,"green")
                                 }
                                 case "Auxr+" => {
drawP2Tcourbe(findCoordXOfPlace(elm.idplace,pn.SetOfPlaces)-radius,
findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)-radius/6,
findCoordXOfTrans(elm.idtrans,pn.SetOfTrans)+findWidthOfTrans(elm.idtrans,pn.SetOfTrans)/2,
findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)+(findCoordYOfTrans(elm.idtrans,pn.SetOfTrans)-
findCoordYOfPlace(elm.idplace,pn.SetOfPlaces))/3,findCoordXOfTrans(elm.idtrans,pn.SetOfTrans)+
findWidthOfTrans(elm.idtrans,pn.SetOfTrans)/2,
findCoordYOfTrans(elm.idtrans,pn.SetOfTrans)-radius,"green")
                                 }
                                 case "Auxr" => {
drawP2Tcourbe(findCoordXOfPlace(elm.idplace,pn.SetOfPlaces)-radius,
findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)-radius/6,
findCoordXOfTrans(elm.idtrans,pn.SetOfTrans)+ findWidthOfTrans(elm.idtrans,pn.SetOfTrans)/2,
findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)+(findCoordYOfTrans(elm.idtrans,pn.SetOfTrans)-
findCoordYOfPlace(elm.idplace,pn.SetOfPlaces))/3,
findCoordXOfTrans(elm.idtrans,pn.SetOfTrans)+findWidthOfTrans(elm.idtrans,pn.SetOfTrans)/2,
findCoordYOfTrans(elm.idtrans,pn.SetOfTrans)-radius,"green")
                                 }
case "In" => {
drawP2Tcourbe(findCoordXOfPlace(elm.idplace,pn.SetOfPlaces),
findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)+radius,
findCoordXOfTrans(elm.idtrans,pn.SetOfTrans) + findWidthOfTrans(elm.idtrans,pn.SetOfTrans)/2,
findCoordYOfTrans(elm.idtrans,pn.SetOfTrans)/2,findCoordXOfTrans(elm.idtrans,pn.SetOfTrans)+
findWidthOfTrans(elm.idtrans,pn.SetOfTrans)/2,
findCoordYOfTrans(elm.idtrans,pn.SetOfTrans)-radius,"black")
                                 }
                                 case _ => {
// for out place - drawing depending from the relative position between place and transition
  if(findCoordXOfPlace(elm.idplace,pn.SetOfPlaces)
```

```
                                        < findCoordXOfTrans(elm.idtrans,pn.SetOfTrans)+
    findWidthOfTrans(elm.idtrans,pn.SetOfTrans)/2) { // ifplace on left of transition
drawP2Tcourbe(findCoordXOfPlace(elm.idplace,pn.SetOfPlaces)+7*radius/10,
findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)+7*radius/10,
findCoordXOfTrans(elm.idtrans,pn.SetOfTrans),
(findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)+(findCoordYOfTrans(elm.idtrans,pn.SetOfTrans)-
findCoordYOfPlace(elm.idplace,pn.SetOfPlaces))/2),findCoordXOfTrans(elm.idtrans,pn.SetOfTrans)+
findWidthOfTrans(elm.idtrans,pn.SetOfTrans)/4,
findCoordYOfTrans(elm.idtrans,pn.SetOfTrans)-5*radius/6,"black")
  } else { // if place on right of transition
if(findCoordXOfPlace(elm.idplace,pn.SetOfPlaces) > findCoordXOfTrans(elm.idtrans,pn.SetOfTrans)+
findWidthOfTrans(elm.idtrans,pn.SetOfTrans)/2) {
 drawP2Tcourbe(findCoordXOfPlace(elm.idplace,pn.SetOfPlaces)-7*radius/10,
 findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)+7*radius/10,
 findCoordXOfTrans(elm.idtrans,pn.SetOfTrans)+findWidthOfTrans(elm.idtrans,pn.SetOfTrans),
 (findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)+(findCoordYOfTrans(elm.idtrans,pn.SetOfTrans)-
 findCoordYOfPlace(elm.idplace,pn.SetOfPlaces))/2),
 findCoordXOfTrans(elm.idtrans,pn.SetOfTrans) +
 3*findWidthOfTrans(elm.idtrans,pn.SetOfTrans)/4,
 findCoordYOfTrans(elm.idtrans,pn.SetOfTrans)-5*radius/6,"black")
   } else { // place and transition aligned
drawP2T(findCoordXOfPlace(elm.idplace,pn.SetOfPlaces),
findCoordYOfPlace(elm.idplace,pn.SetOfPlaces),
findCoordXOfTrans(elm.idtrans,pn.SetOfTrans) + findWidthOfTrans(elm.idtrans,pn.SetOfTrans)/2,
findCoordYOfTrans(elm.idtrans,pn.SetOfTrans)-radius)
                                                    }
                                            }
                                    }
                            }
                    })
// drawing of the arrows of the post_conditions - from transition to place
// for auxiliary places, the orientation of the curvature depends on the
//relative distance between the transition and the place

pn.SetOfPost.foreach( elm => { // for every post_conditions of final Petri Net
 (elm.idplace).head match { // drawing determined by type of pointed place
  case "Auxl" => {
   if((findCoordYOfTrans(elm.idtrans,pn.SetOfTrans)-
     findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)) > 300) {
   // if relative distance is bigger than an arbitrary distance of 300
     drawT2Pcourbe(findCoordXOfTrans(elm.idtrans,pn.SetOfTrans),
     findCoordYOfTrans(elm.idtrans,pn.SetOfTrans)+findHeightOfTrans(elm.idtrans,
           pn.SetOfTrans)/2,findCoordXOfPlace(elm.idplace,pn.SetOfPlaces),
     findCoordYOfTrans(elm.idtrans,pn.SetOfTrans),findCoordXOfPlace(elm.idplace,
           pn.SetOfPlaces),findCoordYOfPlace(elm.idplace,
           pn.SetOfPlaces)+2*radius,"green")
   } else {
     drawT2Pcourbe(findCoordXOfTrans(elm.idtrans,pn.SetOfTrans)+
      findWidthOfTrans(elm.idtrans,pn.SetOfTrans),
     findCoordYOfTrans(elm.idtrans,pn.SetOfTrans),
     findCoordXOfTrans(elm.idtrans,pn.SetOfTrans),
     findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)-9*radius,
     findCoordXOfPlace(elm.idplace,pn.SetOfPlaces)+2*radius,
     findCoordYOfPlace(elm.idplace,pn.SetOfPlaces),"green")
                                        }
```

```
                                    }
    case "Auxl+" => {
     if((findCoordYOfTrans(elm.idtrans,pn.SetOfTrans)-
      findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)) > 300) {
       drawT2Pcourbe(findCoordXOfTrans(elm.idtrans,pn.SetOfTrans),
       findCoordYOfTrans(elm.idtrans,pn.SetOfTrans) +
        findHeightOfTrans(elm.idtrans,pn.SetOfTrans)/2,
       findCoordXOfPlace(elm.idplace,pn.SetOfPlaces),
       findCoordYOfTrans(elm.idtrans,pn.SetOfTrans),
       findCoordXOfPlace(elm.idplace,pn.SetOfPlaces),
       findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)+2*radius,"green")
     } else {
       drawT2Pcourbe(findCoordXOfTrans(elm.idtrans,pn.SetOfTrans) +
        findWidthOfTrans(elm.idtrans,pn.SetOfTrans)/2,
       findCoordYOfTrans(elm.idtrans,pn.SetOfTrans)-radius,
       findCoordXOfTrans(elm.idtrans,pn.SetOfTrans)+findWidthOfTrans(elm.idtrans,
        pn.SetOfTrans)/2,findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)+
        (findCoordYOfTrans(elm.idtrans,pn.SetOfTrans)-
        findCoordYOfPlace(elm.idplace,pn.SetOfPlaces))/3,
       findCoordXOfPlace(elm.idplace,pn.SetOfPlaces)+2*radius,
       findCoordYOfPlace(elm.idplace,pn.SetOfPlaces),"green")
                                        }
                                    }
    case "Auxr" => {
     if((findCoordYOfTrans(elm.idtrans,pn.SetOfTrans)-
      findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)) > 300) {
       drawT2Pcourbe(findCoordXOfTrans(elm.idtrans,pn.SetOfTrans)+
        findWidthOfTrans(elm.idtrans,pn.SetOfTrans),findCoordYOfTrans(elm.idtrans,
         pn.SetOfTrans) + findHeightOfTrans(elm.idtrans,pn.SetOfTrans)/2,
       findCoordXOfPlace(elm.idplace,pn.SetOfPlaces),
       findCoordYOfTrans(elm.idtrans,pn.SetOfTrans),
       findCoordXOfPlace(elm.idplace,pn.SetOfPlaces),
       findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)+2*radius,"green")
     } else {
      drawT2Pcourbe(findCoordXOfTrans(elm.idtrans,pn.SetOfTrans)+
       findWidthOfTrans(elm.idtrans,pn.SetOfTrans)/2,findCoordYOfTrans(elm.idtrans,
        pn.SetOfTrans),findCoordXOfTrans(elm.idtrans,pn.SetOfTrans),
       findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)-9*radius,
       findCoordXOfPlace(elm.idplace,pn.SetOfPlaces)-2*radius,
       findCoordYOfPlace(elm.idplace,pn.SetOfPlaces),"green")
                                        }
                                    }
case "Auxr+" => {
 if((findCoordYOfTrans(elm.idtrans,pn.SetOfTrans)-
       findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)) > 300) {
  drawT2Pcourbe(findCoordXOfTrans(elm.idtrans,pn.SetOfTrans)+
   findWidthOfTrans(elm.idtrans,pn.SetOfTrans),
   findCoordYOfTrans(elm.idtrans,pn.SetOfTrans) +
    findHeightOfTrans(elm.idtrans,pn.SetOfTrans)/2,
   findCoordXOfPlace(elm.idplace,pn.SetOfPlaces),
   findCoordYOfTrans(elm.idtrans,pn.SetOfTrans),
   findCoordXOfPlace(elm.idplace,pn.SetOfPlaces),
   findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)+2*radius,"green")
 } else {
  drawT2Pcourbe(findCoordXOfTrans(elm.idtrans,pn.SetOfTrans) +
```

```
    findWidthOfTrans(elm.idtrans,pn.SetOfTrans)/2,
    findCoordYOfTrans(elm.idtrans,pn.SetOfTrans)-radius,
    findCoordXOfTrans(elm.idtrans,pn.SetOfTrans) +
     findWidthOfTrans(elm.idtrans,pn.SetOfTrans)/2,
    findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)+
     (findCoordYOfTrans(elm.idtrans,pn.SetOfTrans)-
     findCoordYOfPlace(elm.idplace,pn.SetOfPlaces))/3,
    findCoordXOfPlace(elm.idplace,pn.SetOfPlaces)-2*radius,
    findCoordYOfPlace(elm.idplace,pn.SetOfPlaces),"green")
                                          }
                                    }
case _ => { //for other places drawing of straight lines
 if(findCoordXOfTrans(elm.idtrans,pn.SetOfTrans) +
  findWidthOfTrans(elm.idtrans,pn.SetOfTrans)/2 <
   findCoordXOfPlace(elm.idplace,pn.SetOfPlaces)) {
       // line tilted from left to right
    drawT2P(findCoordXOfTrans(elm.idtrans,pn.SetOfTrans) +
     findWidthOfTrans(elm.idtrans,pn.SetOfTrans)/2,
    findCoordYOfTrans(elm.idtrans,pn.SetOfTrans) +
     findHeightOfTrans(elm.idtrans,pn.SetOfTrans),
    findCoordXOfPlace(elm.idplace,pn.SetOfPlaces)-15*radius/10,
    findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)-radius)
 } else { // line tilted from right to  left
 if(findCoordXOfTrans(elm.idtrans,pn.SetOfTrans) +
  findWidthOfTrans(elm.idtrans,pn.SetOfTrans)/2 >
   findCoordXOfPlace(elm.idplace,pn.SetOfPlaces)) {
    drawT2P(findCoordXOfTrans(elm.idtrans,pn.SetOfTrans) +
     findWidthOfTrans(elm.idtrans,pn.SetOfTrans)/2,
    findCoordYOfTrans(elm.idtrans,pn.SetOfTrans) +
     findHeightOfTrans(elm.idtrans,pn.SetOfTrans),
    findCoordXOfPlace(elm.idplace,pn.SetOfPlaces)+15*radius/10,
    findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)-radius)
 } else { // same x position, the straight line is vertical
  drawT2P(findCoordXOfTrans(elm.idtrans,pn.SetOfTrans) +
   findWidthOfTrans(elm.idtrans,pn.SetOfTrans)/2,
   findCoordYOfTrans(elm.idtrans,pn.SetOfTrans) +
   findHeightOfTrans(elm.idtrans,pn.SetOfTrans),
   findCoordXOfPlace(elm.idplace,pn.SetOfPlaces),
   findCoordYOfPlace(elm.idplace,pn.SetOfPlaces)-2*radius)
                                                }
                                          }
                                    }
                              }
                  })
// drawing of the transitions in black, with the exception
// of the firable listed in the mapping, that are drawn in red

  pn.SetOfTrans.foreach (elm => {
    // if NOT belongToMap, draw in black
  if(belongToMap(mp,elm.idtrans)._2 == Nil) {
   if(elm.token == "") {
    drawTransition(elm.tx,elm.ty,elm.width,elm.height)
    drawTransName(elm.tx + 18,elm.ty,elm.width,elm.height,elm.name)
  } else {
    drawTransition(elm.tx,elm.ty,elm.width,elm.height)
```

```
      drawText(elm.tx + 5,elm.ty,elm.width,elm.height,elm.name,
       elm.token,elm.density)
    }
   } else { // else draw in colour
    if(elm.token == "") {
     drawTransitionColour(elm.tx,elm.ty,elm.width,elm.height)
     drawTransName(elm.tx + 18,elm.ty,elm.width,elm.height,elm.name)
    } else {
     drawTransitionColour(elm.tx,elm.ty,elm.width,elm.height)
     drawText(elm.tx + 5,elm.ty,elm.width,elm.height,elm.name,
                  elm.token,elm.density)
                                    }
                            }
                   })

// drawing of the places and their number of tokens

    pn.SetOfPlaces.foreach( elm => {
     (elm.idplace).head match {
      case "In" => {
       drawPlace(elm.cx,elm.cy+radius,elm.radius,"blue")
       drawNbrTok(elm.cx,elm.cy+radius,elm.nbrTokens)
      }
      case "Auxl" => {
       drawPlace(elm.cx,elm.cy,elm.radius,"green")
       drawNbrTok(elm.cx,elm.cy,elm.nbrTokens)
      }
      case "Auxr" => {
       drawPlace(elm.cx,elm.cy,elm.radius,"green")
       drawNbrTok(elm.cx,elm.cy,elm.nbrTokens)
      }
case "Auxl+" => {
 drawPlace(elm.cx,elm.cy,elm.radius,"green")
 drawNbrTok(elm.cx,elm.cy,elm.nbrTokens)
}
case "Auxr+" => {
 drawPlace(elm.cx,elm.cy,elm.radius,"green")
 drawNbrTok(elm.cx,elm.cy,elm.nbrTokens)
}
case "Out" => {
   // final out of the Petri Net in blue
 if ((elm.idplace).tail == Nil) {
  drawPlace(elm.cx,elm.cy,elm.radius,"blue")
  drawNbrTok(elm.cx,elm.cy,elm.nbrTokens)
 } else { // internal out of the Petri Net in black
  drawPlace(elm.cx,elm.cy,elm.radius,"black")
  drawNbrTok(elm.cx,elm.cy,elm.nbrTokens)
 }
}
case "ns" => {
 if (elm.name != "") {
  drawTokenName(xOfIn + 3*radius/2,yOfIn,elm.name)
  drawPlace(xOfIn + 7*radius/2,yOfIn,radius,"black")
  drawNbrTok(xOfIn + 7*radius/2,yOfIn,elm.nbrTokens)
 }
```

```
}
case "as" => {
 if(elm.name != "") {
  drawPlace(xOfIn + 13*radius/2, yOfIn,radius,"red")
  drawNbrTok(xOfIn + 13*radius/2,yOfIn,elm.nbrTokens)
  yOfIn += 3*radius
                                                }
                                            }
                                        }
                                    })
endSVG()
close_file()
        }

}
```

Appendix I

Running the Petri Net

I.1 Running Petri Net

This appendix lists the full code of the sub-procedures used by the procedure *execute* of the class *runningPetriNet*. This procedure executes the Petri Net. Using a pre-selected list of firable transitions, it fires one of them, modifies the number of tokens in the involved places, redraws the new state of the Petri Net, and refreshes the list of firable transitions.

```
class runningPetriNet(var pn : PetriNet) {

 var mapOfFirableTrans     : Map[Int, List[String]] = Map()
                              // Map between number int and firable transition
 var curentPlace           : List[String] = Nil
 var curentTrans           : List[String] = Nil
 var curentListPre         : List[pre_condition] = Nil
 var curentListPost        : List[post_condition] = Nil
 var listIdTrans           : List[List[String]] = Nil
 var listIdTransCopy       : List[List[String]] = Nil
 var curentListExecPre     : List[pre_condition] = Nil
 var curentListPreSpaces   : List[pre_condition] = Nil
 var curentListPostSpaces  : List[post_condition] = Nil

 /* internal methods to retrieve usefull informations
 from lists of the Petri Net */

 def subListPre(idTrans : List[String], lPreCond: List[pre_condition])
                                                        : List[pre_condition] = {
 // return a list of all the pre_conditions corresponding to a specific idTrans
  lPreCond match {
   case Nil => Nil
   case pre_condition(idplace,idtrans,weight,caraGraphi)::l => {
    if (idtrans == idTrans) {
     pre_condition(idplace,idtrans,weight,caraGraphi)::subListPre(idTrans,l)
    } else {
     subListPre(idTrans,l)
    }
```

```
    }
   }
  }

  def subListPreSpaces(idTrans : List[String], lPreSpaces : List[pre_condition]) :
                                                          List[pre_condition] = {
// return a list of pre_conditions (related to the token space) for a
// specific idTrans
   lPreSpaces match {
    case Nil => Nil
    case pre_condition(idplace,idtrans,weight,caraGraphi)::l => {
     if (idtrans == idTrans) {
      pre_condition(idplace,idtrans,weight,caraGraphi)::subListPreSpaces(idTrans,l)
     } else {
      subListPreSpaces(idTrans,l)
     }
    }
   }
  }

  def subListPost(idTrans : List[String], lPostCond: List[post_condition]) :
                                                        List[post_condition] = {
      // return a list of post_condition corresponding to a specific idTrans
   lPostCond match {
    case Nil => Nil
    case post_condition(idtrans,idplace,weight,caraGraphi)::l => {
     if (idtrans == idTrans) {
      post_condition(idtrans,idplace,weight,caraGraphi)::subListPost(idTrans,l)
     } else {
      subListPost(idTrans,l)
     }
    }
   }
  }

  def subListPostSpaces(idTrans : List[String], lPostSpaces : List[post_condition]) :
                                                            List[post_condition] = {
     // return a list of post_condition (related to the token space)
     // for a specific idTrans
   lPostSpaces match {
    case Nil => Nil
    case post_condition(idtrans,idplace,weight,caraGraphi)::l => {
     if (idtrans == idTrans) {
      post_condition(idtrans,idplace,weight,caraGraphi)::subListPostSpaces(idTrans,l)
     } else {
      subListPostSpaces(idTrans,l)
     }
    }
   }
  }

  def getNumbTokInPlace(idPlace : List[String], lplace : List[place]) : Int = {
    // return the number of tokens for a specific idPlace, from a list of places
   lplace match {
    case Nil => 0
```

```
  case place(cx,cy,radius,entry,exit,regular,name,nbrTokens,nbrReq,idplace,
                                                              caraGraphi)::l => {
   if (idplace == idPlace) {
              nbrTokens
   } else {
              getNumbTokInPlace(idPlace,l)
   }
  }
 }
}

def firablePre(idTrans : List[String], pre : List[pre_condition], pla :
                                    List[place]) : Boolean = {
  // check if a specific transition (idTrans), for all its related places,
  // is firable or not

 var lTransPre   : List[pre_condition] = Nil
 var lPlaceNames : List[List[String]] = Nil
 var lPlaces     : List[place] = Nil
 var result      : Boolean = true

 lTransPre = subListPre(idTrans,pre)

 lTransPre.foreach(w => {
              //println(w.weight,getNumbTokInPlace(w.idplace,pla))
 if((result == true) && (w.weight <= getNumbTokInPlace(w.idplace,pla))) {
              result = result & true
 } else {
              result = false
 }
 })

return result
}

def firablePreSpaces(idTrans : List[String], pre : List[pre_condition],
                                        pla : List[place]) : Boolean = {
  // check if a specific transition (idTrans),
  // for all its related places in the token space, is firable or not

 var lTransPre : List[pre_condition] = Nil
 var lPlaces   : List[place] = Nil
 var result    : Boolean = true

 lTransPre = subListPreSpaces(idTrans,pre)

 lTransPre.foreach(w => {
   //println(w.weight,getNumbTokInPlace(w.idplace,pla))
 if((result == true) && (w.weight <= getNumbTokInPlace(w.idplace,pla))) {
                     result = result & true
 } else {
                     result = false
 }
 })
```

```
 return result
}

def choiceFirableTrans(mapChoice : Map[Int, List[String]]) : List[String] = {
   // return the choice among a mapping of firable transitions

 var choice   : Int = 0
 var response : List[String] = Nil

 println("List of the firable transitions:\n")
 mapChoice.foreach(elm => {println(elm,"\n")})

 while(response == Nil) {
       println("Make your choice:\n")
       var s = (scala.io.StdIn.readLine()).toInt
       println(s)
       if (mapChoice contains s) {
                     response = mapChoice(s)
       } else {
                     println("Value not present.\n")
       }
 }

 return response
}

def getNameInTrans(idTrans: List[String], lTrans : List[transition]) : String = {
   // return the name of a specific idTrans among a list of transitions
 lTrans match {
 case Nil => ""
 case transition(tx,ty,width,height,name,token,density,idtrans,caraGraphi)::l => {
  if(idtrans == idTrans) {
               name
  } else {

               getNameInTrans(idTrans,l)
  }
 }
 }
}

def substract(a:Int,b:Int) : Int = { // compute difference between a and b
               a - b
}

def add(a:Int,b:Int) : Int = { // compute summation of a and b
               a + b
}

def substractInPlace(idPlace : List[String],lplace : List[place],
                                    weight : Int) : List[place] = {
 // substract weight (of a transition) from number of tokens in a
 // specific idPlace
 lplace match {
  case Nil => Nil
```

```
  case place(cx,cy,radius,entry,exit,name,regular,nbrTokens,nbrReq,
                                                idplace,caraGraphi)::l => {
   if (idplace == idPlace) {
    place(cx,cy,radius,entry,exit,name,regular,
                        substract(getNumbTokInPlace(idPlace,lplace),
     weight),nbrReq,idplace,caraGraphi)::substractInPlace(idPlace,l,weight)
   } else {
    place(cx,cy,radius,entry,exit,name,regular,nbrTokens,
        nbrReq,idplace,caraGraphi) ::substractInPlace(idPlace,l,weight)
   }
  }
 }
}

def addInPlace(idPlace : List[String],lplace : List[place],
                                              weight : Int) : List[place] = {
 // add weight (of a transition) to number of tokens in a specific idPlace
 lplace match {
  case Nil => Nil
  case place(cx,cy,radius,entry,exit,name,regular,nbrTokens,nbrReq,idplace,
                                                          caraGraphi)::l => {
   if (idplace == idPlace) {
    place(cx,cy,radius,entry,exit,name,regular,
       add(getNumbTokInPlace(idPlace,lplace),weight)*nbrReq,nbrReq,idplace,
                                    caraGraphi)::addInPlace(idPlace,l,weight)
   } else {
    place(cx,cy,radius,entry,exit,name,regular,nbrTokens,nbrReq,
       idplace,caraGraphi)::addInPlace(idPlace,l,weight)
   }
  }
 }
}

/* end of internal usefull methods */

/* printing methods */

def printList(lplace : List[place]) {
 lplace.foreach( x => {
       println(x)
 })
}

def printListPre(lpre : List[pre_condition]) {
 lpre.foreach( x => {
       println(x)
 })
}

def printListPost(lpost : List[post_condition]) {
 lpost.foreach( x => {
       println(x)
 })
}
```

```
 /* end of printing methods */
```

I.2 Running Petri Net Main Methods

```
def modifyInPetriNet(idTrans : List[String], pn : PetriNet) : PetriNet = {
  // modify number of token in places after firing of transitions

 var lTransPre         : List[pre_condition]  = Nil
 var lTransPreSpaces   : List[pre_condition]  = Nil
 var lTransPost        : List[post_condition] = Nil
 var lTransPostSpaces  : List[post_condition] = Nil
 var lPlaces           : List[place]          = Nil
 var poids             : Int                  = 0
 var idplace           : List[String]         = Nil
 var nbrTok            : Int                  = 0
 var setAux            : List[place]          = pn.SetOfPlaces

    // following type of transition

 getNameInTrans(idTrans,pn.SetOfTrans) match {
    // transition concerned by Petri Net and token space
  case "tell" | "get" | "ask" | "nask" => {
      //  collect all the pre of a specific idTrans
   lTransPre = subListPre(idTrans,pn.SetOfPre)
      // collect all the pre of a specific idTrans related to token space
   lTransPreSpaces = subListPreSpaces(idTrans,pn.SetOfPreSpaces)
      // collect all the post of a specific idTrans
   lTransPost = subListPost(idTrans,pn.SetOfPost)
      // collect all the post of a specific idTrans related to token space
   lTransPostSpaces = subListPostSpaces(idTrans,pn.SetOfPostSpaces)

   lTransPre.foreach( x => { // for every pre ...
       // substract weight of place idplace
    setAux = substractInPlace(x.idplace,setAux,x.weight)
   })
lTransPost.foreach( x => { // for every post ...
       // add weight of place idplace
    setAux = addInPlace(x.idplace,setAux,x.weight)
   })
       // for every pre of token space ...
   lTransPreSpaces.foreach( x => {
       // substract weight
    setAux = substractInPlace(x.idplace,setAux,x.weight)
   })
       // for every post of token space ...
   lTransPostSpaces.foreach( x => {
       // add weight of place idplace
    setAux =  addInPlace(x.idplace,setAux,x.weight)
   })

     // convert adapted Petri Net to svg
   pn2Svg.convertPn2Svg(PetriNet(setAux,pn.SetOfTrans,
                      pn.SetOfPre,pn.SetOfPost,
```

```
            pn.SetOfPreSpaces,pn.SetOfPostSpaces),Map())
    return PetriNet(setAux,pn.SetOfTrans,pn.SetOfPre,
      pn.SetOfPost,pn.SetOfPreSpaces,pn.SetOfPostSpaces)

  }
     // transition concerned by Petri Net only
  case _       => {
   lTransPre = subListPre(idTrans,pn.SetOfPre)
   lTransPost = subListPost(idTrans,pn.SetOfPost)

   lTransPre.foreach( x => {
    setAux = substractInPlace(x.idplace,setAux,x.weight)
   })

   lTransPost.foreach( x => {
    setAux = addInPlace(x.idplace,setAux,x.weight)
   })

   pn2Svg.convertPn2Svg(PetriNet(setAux,pn.SetOfTrans,
              pn.SetOfPre,pn.SetOfPost,
           pn.SetOfPreSpaces,pn.SetOfPostSpaces),Map())
   return PetriNet(setAux,pn.SetOfTrans,pn.SetOfPre,
     pn.SetOfPost,pn.SetOfPreSpaces,pn.SetOfPostSpaces)
  }
 }
}
/* end of method for modifying the petri Net */

 /* method for constructing a map of firable transitions */
 /* scan of all the transitions, and with their related places,
    check of their firable character with regard to */
 /* the concerned pre_conditions related to the Petri Net from
    one part and its extension with token space on   */
 /* the other part  */
 def constructMapOfFirableTrans(lTrans : List[transition],
  lPlace : List[place],
  lPre : List[pre_condition],lPreSpaces : List[pre_condition])
    : Map[Int, List[String]] = {

   // searching of firable transitions (following their type)
   // and building a map with the positive one

   var i : Int = 1
   var mapOfFirableTransL : Map[Int, List[String]] = Map()
   var firable : Boolean = false
     // for every transition of final Petri Net
   lTrans.foreach(elm => {
     // if transition concerned by Petri Net AND
     // its extension by token space
    elm.name match {
       // evaluate if firable in both cases
     case "tell" | "get" | "ask" | "nask" => {
      if(firablePre(elm.idtrans,lPre,lPlace) &&
         firablePreSpaces(elm.idtrans,lPreSpaces,lPlace)) {
              mapOfFirableTransL(i) = elm.idtrans
```

```
                firable = true
     }
    }
     // for the others, evaluate if firable only with
     // regard to Petri Net
    case _ => {
     if(firablePre(elm.idtrans,lPre,lPlace)) {
      mapOfFirableTransL(i) = elm.idtrans
                firable = true
    }
   }
  } // match
  if (firable) {
    i = i + 1
    firable = false
  }
  })
 return mapOfFirableTransL
}
/* end of method for constructing a map
                    of firable transitions */
/* main method for renewing the Petri Net
              after firing of a transition */
/* while the list of firable transitions is not empty,
   select one of these transitions and fire it */
/* modify the Petri Net with respect of the pre-
  and post-conditons or this fired transition       */
/* redraw the modified Petri Net */
/* reset the list of firable transitions except if
   the final Out place is reached (its number of   */
/* tokens is different from 0 */

def execute(pn : PetriNet) {

 var pnAux : PetriNet = pn
 mapOfFirableTrans = constructMapOfFirableTrans(
            pnAux.SetOfTrans,pnAux.SetOfPlaces,
            pnAux.SetOfPre,pnAux.SetOfPreSpaces)
 pn2Svg.apply(pnAux,mapOfFirableTrans)

 while (! mapOfFirableTrans.isEmpty) {
  curentTrans = choiceFirableTrans(mapOfFirableTrans)
  println("Curent chosen transition :")
  println(curentTrans)
  pnAux = modifyInPetriNet(curentTrans,pnAux)
  if(getNumbTokInPlace("Out"::Nil,pnAux.SetOfPlaces) == 0) {
   mapOfFirableTrans = constructMapOfFirableTrans(
           pnAux.SetOfTrans,pnAux.SetOfPlaces,
              pnAux.SetOfPre,pnAux.SetOfPreSpaces)
   pn2Svg.apply(pnAux,mapOfFirableTrans)
  } else {
   mapOfFirableTrans = Map()
   pn2Svg.apply(pnAux,mapOfFirableTrans)
  }
 }
```

```
}
```

Part VI

References

Bibliography

[Arb96] F. Arbab. The IWIM Model for Coordination of Concurrent Activities. In P. Ciancarini and C. Hankin, editors, *Proceedings of the First International Conference on Coordination Models and Languages*, volume 1061, pages 34–56, Cesena, Italy, 1996. Springer-Verlag. 29, 40

[Arb04] Farhad Arbab. Reo: A channel-based coordination model for component composition. *Mathematical. Structures in Computer Science.*, 14(3):329–366, June 2004. 30, 40

[BBG09] P. Baldan, F. Bonchi, and F. Gadducci. Encoding asynchronous interactions using open petri nets. In *CONCUR 2009 - Concurrency Theory, 20th International Conference, CONCUR 2009, Bologna, Italy, September 1-4, 2009. Proceedings*, pages 99–114, 2009. 287, 289, 544

[BCGZ01] N. Busi, P. Ciancarini, R. Gorrieri, and G. Zavattaro. Coordination Models: A Guided Tour. In Andra Omicini, Franco Zambonelli, Matthias Klush, and Robert Tolksdorf, editors, *Coordination of Internet Agents: Models, Technologies, and Applications*, pages 6–24. Springer-Verlag, 2001. 3, 27, 39

[BCP07] D. Balzarotti, P. Costa, and G. P. Picco. The lights tuple space framework and its customization for context-aware applications. *Web Intelligence and Agent Systems*, 5(2):215–231, 2007. 221

[BGLG05] M. Bravetti, R. Gorrieri, R. Lucchi, and G.Zavattaro. Quantitative Information in the Tuple Space Coordination Model. *Theoretical Computer Science*, 346(1):28–57, 2005. 37, 40, 82

[BGLZ04] M. Bravetti, R. Gorrieri, R. Lucchi, and G. Zavattaro. Probabilistic and Prioritized Data Retrieval in the Linda Coordination Model. In R. De Nicola, G.L. Ferrari, and G. Meredith, editors, *Proceedings of the 6th International Conference on Coordination Models and Languages*, volume 2949 of *Lecture Notes in Computer Science*, pages 55–70. Springer, 2004. 37, 40, 82

[BGZ] N. Busi, R. Gorrieri, and G. Zavattaro. A Process Algebraic View of Linda Coordination Primitives. 82

[BGZ97] N. Busi, R. Gorrieri, and G. Zavattaro. On the Turing equivalence of Linda coordination primitives. *Electronic Notes in Theoretical Computer Science*, 7:75–75, 1997. 82

[BJ93] K.D. Bosschere and J.-M. Jacquet. Multi-prolog: Definition, operational semantics and implementation. In *Logic Programming, Proceedings of the Tenth International Conference on Logic Programming, Budapest, Hungary, June 21-25, 1993*, pages 299–313, 1993. 221

[BJ98] A. Brogi and J.-M. Jacquet. On the Expressiveness of Linda-like Concurrent Languages. *Electronic Notes in Theoretical Computer Science*, 16(2):61–82, 1998. 49, 50, 51, 54, 55, 56, 57, 68, 69, 97, 130, 134, 138, 153, 155, 156, 162, 377, 378, 379, 380, 385, 386, 388, 394

[BJ99] A. Brogi and J.-M. Jacquet. On the Expressiveness of Coordination Models. In C. Ciancarini and A. Wolf, editors, *Proceedings of the Third International Conference on Coordination Languages and Models*, volume 1594 of *Lecture Notes in Computer Science*, pages 134–149. Springer-Verlag, Apr 1999. 48, 57, 64, 68, 69, 377, 382, 384, 386, 388, 389, 391, 395

[BJ03a] A. Brogi and J.-M. Jacquet, editors. *Foclasa 2002, Foundations of Coordination Languages and Software Architectures (Satellite Workshop of CONCUR 2002)*, volume 68, 2003. 26, 48, 49, 50, 51, 68, 69, 377

[BJ03b] A. Brogi and J.-M. Jacquet. On the Expressiveness of Coordination via Shared Dataspaces. *Science of Computer Programming*, 46(1-2):71–98, 2003. 26, 48, 49, 50, 51, 57, 61, 64, 69, 72, 125, 130, 163, 167, 382, 384, 386, 388, 389, 391, 392, 393, 395, 396

[BJK06] A. Brogi, J.-M. Jacquet, and J. Kok. Foundations of Coordination Languages and Software Architectures. *Fundamenta Informaticae*, 73(4):431–598, 2006. 26

[BJKP06] A. Brogi, J.-M. Jacquet, J. Kramer, and E. Pimentel. Second International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'03). *Science of Computer Programming*, 61(2):73–187, 2006. 26

[BJL06] A. Brogi, J.-M. Jacquet, and I. Linden. Fully Abstract Semantics for a Coordination Model with Asynchronous Communication and Enhanced Matching. *Fundamentae Informatica*, 73(4):431–478, 2006. 75, 374

[BJP04] A. Brogi, J.-M. Jacquet, and E. Pimentel, editors. *Proceedings of FOCLASA 2003, the Foundations of Coordination Languages and Software Architectures, a satellite event of CONCUR 2003*, volume 97 of *Electronic Notes in Theoretical Computer Science*, 2004. 26

[BKZ99] M. M. Bonsangue, J. N. Kok, and G. Zavattaro. Comparing coordination models based on shared distributed replicated data. In *ACM Symposium on Applied Computing*, pages 156–165, 1999. 82

[BL93] J.-P Banâtre and D. Le Métayer. Programming by multiset transformation. *Communications of the ACM*, 36(1):98–111, 1993. 32, 40

[BM96] J.-P. Banâtre and D. Le Métayer. Gamma and the Chemical Reaction Model: Ten Years After. *Coordination Programming*, pages 3–41, Imperial College Press, London, 1996. 32, 40

[CA10] Dave Clarke and Gul A. Agha, editors. *Coordination Models and Languages, 12th International Conference, COORDINATION 2010, Amsterdam, The Netherlands, June 7-9, 2010. Proceedings*, volume 6116 of *Lecture Notes in Computer Science*. Springer, 2010. 26

[CDH00] Proceedings of the 2000 ACM Symposium on Applied Computing (SAC 2000). pages 147–282, Como (I), 19–21 March 2000. ACM. Track on Coordination Models, Languages and Applications. 26

[CG89] Nicholas Carriero and David Gelernter. Linda in Context. *Commun. ACM*, 32(4):444–458, 1989. 27

[CGZ95] N. Carriero, D. Gelernter, and L. Zuck. Bauhaus linda. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems: Proc. of the ECOOP'94 Workshop on Modles and Languages for Coordination of Parallelism and Distribution*, pages 66–76. Springer, Berlin, Heidelberg, 1995. 28

[CP15] Javier Cámara and José Proença, editors. *Proceedings 13th International Workshop on Foundations of Coordination Languages and Self-Adaptive Systems, FOCLASA 2014, Rome, Italy, 6th September 2014*, volume 175 of *EPTCS*, 2015. 26

[CV06] C. Canal and M. Viroli, editors. *Proceedings of the 4th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2005)*, volume 154 of *Electronic Notes in Theoretical Computer Science*, 2006. 26

[CV13] Carlos Canal and Massimo Villari, editors. *Advances in Service-Oriented and Cloud Computing - Workshops of ESOCC 2013, Málaga, Spain, September 11-13, 2013, Revised Selected Papers*, volume 393 of *Communications in Computer and Information Science*. Springer, 2013. 26

[CW006] Coordination models and languages, 8th international conference, coordination 2006. volume 4038 of *Lecture Notes in Computer Science*, Bologna, Italy, 2006. Springer-Verlag, 2006. 26

[CW111] *SAC 2011: Proceedings of the 2011 ACM Symposium on Applied Computing*, New York, NY, USA, 2011. ACM. Special Track on Coordination Models, Languages and Applications. 26

[dBP94] F.S. de Boer and C. Palamidessi. Embedding as a Tool for Language Comparison. *Information and Computation*, 108(1):128–157, 1994. 49, 50, 68, 95, 128, 131

[DJL13a] D. Darquennes, J.-M. Jacquet, and I. Linden. On Density in Coordiantion Languages. In C. Canal and M. Villari, editors, *CCIS 393, Advances in Service-Oriented and Cloud Computing, ESOCC 2013, Proceedings of Foclasa Workshop*, pages 189–203, Malaga, Spain, 2013. Springer. 20, 133

[DJL13b] D. Darquennes, J.-M. Jacquet, and I. Linden. On the Introduction of Density in Tuple-Space Coordination Languages. In *Science of Computer Programming, special issue of Foclasa 2013*. Springer, 2013. 20

[DJL14] D. Darquennes, J.-M. Jacquet, and I. Linden. On Distributed Density in Tuple-based Coordination Languages. In *CONCUR 2014, Proceedings of Foclasa Workshop*, Rome, Italy, 2014. Springer. 21

[FHA99] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces: Principles, Patterns, and Practice*. Addison-Wesley, 1999. 28

[GC92] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, 1992. 23, 39

[Gel85] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985. 24, 40

[GFM04] R. De Nicola G. Ferrari and G. Meredith, editors. *Coordination Languages and Models, 6th International Conference, COORDINATION 2004, Pisa, Italy, February 2004, Proceedings*, volume 2949 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004. 26

[GPR99] A.L. Murphy G.P. Picco and G.-C. Roman. LIME: Linda Meets Mobility. In D. Garlan, editor, *ICSE'99: Proceedings of the 21st International Conference on Software Engineering*, pages 368–377. IEEE Computer Society Press, 1999. 28, 36, 40

[HV15] Tom Holvoet and Mirko Viroli, editors. *Coordination Models and Languages - 17th IFIP WG 6.1 International Conference, COORDINATION 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings*, volume 9037 of *Lecture Notes in Computer Science*. Springer, 2015. 26

[ILJ11] M.-O. Staicu I. Linden and J.-M. Jacquet. On Coordination in Mobile Ad-hoc Networks: Language, Design, Expressiveness, Issues and Semantic Studies. Technical report, Faculty of Computer Science, University of Namur, Namur, Belgium, 2011. 41

[JBB00] J.-M. Jacquet, K. De Bosschere, and A. Brogi. On Timed Coordination Languages. In A. Porto and G.-C. Roman, editors, *Proc. 4th International Conference on Coordination Languages and Models*, volume 1906 of *Lecture Notes in Computer Science*, pages 81–98. Springer, 2000. 41

[JL07] J.-M. Jacquet and I. Linden. Coordinating Context-aware Applications in Mobile Ad-hoc Networks. In T. Braun, D. Konstantas, S. Mascolo, and M. Wulff, editors, *Proceedings of the first ERCIM workshop on eMobility*, pages 107–118. The University of Bern, 2007. 41, 43

[JL09] J.-M. Jacquet and I. Linden. Fully Abstract Models and Refinements as Tools to Compare Agents in Timed Coordination Languages. *Theoretical Computer Science*, 410(2-3):221–253, 2009. 41

[JLD16] J.M. Jacquet, I. Linden, and D. Darquennes. On the introduction of density in tuple-space coordination languages. *Sci. Comput. Program.*, 115-116:149–176, 2016. 26

[JM17] J.-M. Jacquet and Mieke Massink, editors. *Coordination Models and Languages - 19th International Conference, COORDINATION 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchtel, Switzerland, June 19-22, 2017, Proceedings*, Lecture Notes in Computer Science. Springer, 2017. 26

[JP05] J.-M Jacquet and G. P. Picco, editors. *Coordination Models and Languages, 7th International Conference, COORDINATION 2005, Namur, Belgium, April 20-23, 2005, Proceedings*, volume 3454 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005. 26

[KP14] Eva Kühn and Rosario Pugliese, editors. *Coordination Models and Languages - 16th IFIP WG 6.1 International Conference, COORDINATION 2014, Held as Part of the 9th International Federated Conferences on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings*, volume 8459 of *Lecture Notes in Computer Science*. Springer, 2014. 26

[KR12] Natallia Kokash and António Ravara, editors. *Proceedings 11th International Workshop on Foundations of Coordination Languages and Self Adaptation, FOCLASA 2012, Newcastle, U.K., September 8, 2012*, volume 91 of *EPTCS*, 2012. 26

[Lin07] I. Linden. *On the Introduction of Time in Coordination Languages, Semantics, Expressiveness and Programming Methodologies*. PhD thesis, Faculty of Computer Science, University of Namur, Namur, Belgium, 2007. 41, 49, 57, 381

[LJ04] I. Linden and J.-M. Jacquet. On the Expressiveness of Absolute-Time Coordination Languages. In R. De Nicola, G.L. Ferrari, and G. Meredith, editors, *Proc. 6th International Conference on Coordination Models and Languages*, volume 2949 of *Lecture Notes in Computer Science*, pages 232–247. Springer, 2004. 41, 48, 49

[LJ07] I. Linden and J.-M. Jacquet. On the Expressiveness of Timed Coordination via Shared Dataspaces. *Electronical Notes in Theoretical Computer Science*, 180(2):71–89, 2007. 41, 48, 49

[LJBB04] I. Linden, J.-M. Jacquet, K. De Bosschere, and A. Brogi. On the Expressiveness of Relative-Timed Coordination Models. *Electronical Notes in Theoretical Computer Science*, 97:125–153, 2004. 41, 48, 49

[LP16] Alberto Lluch Lafuente and Jos Proena, editors. *Coordination Models and Languages - 18th IFIP International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9686 of *Lecture Notes in Computer Science*. Springer, 2016. 26

[MdM11] P. Matiello and A. C. V. de Melo. A π-calculus Internal Domain-Specific Language for Scala. Technical report, Department of Computer Science, University of São Paulo, São Paulo, Brazil, 2011. 221, 249

[MO06] et al M. Odersky. An Overview of the Scala Programming Language - Technical Report - Second Edition. Technical report, Ecole Polytechnique Federale de Lausanne (EPFL), Lausanne, Switzerland, 2006. 172

[MO10] B. Venners M. Odersky, L. Spoon. *Programming in Scala*. Artima, 2010. 173

[MO13] S. Mariani and A. Omicini. Tuple-based Coordination of Stochastic Systems with Uniform Primitives. In *Proceedings of the 14th Workshop "From Objects to Agents" co-located with the 13th Conference of the Italian Association for Artificial Intelligence (AI*IA 2013), Torino, Italy, December 2-3, 2013.*, pages 8–15, 2013. 38, 40

[MR11a] Wolfgang De Meuter and Gruia-Catalin Roman, editors. *Coordination Models and Languages - 13th International Conference, COORDINATION 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings*, volume 6721 of *Lecture Notes in Computer Science*. Springer, 2011. 26

[MR11b] Mohammad Reza Mousavi and António Ravara, editors. *Proceedings 10th International Workshop on the Foundations of Coordination Languages and Software Architectures, FOCLASA 2011, Aachen, Germany, 10th September, 2011*, volume 58 of *EPTCS*, 2011. 26

[MS10] Mohammad Reza Mousavi and Gwen Salaün, editors. *Proceedings Ninth International Workshop on the Foundations of Coordination Languages and Software Architectures, FOCLASA 2010, Paris, France, 4th September 2010*, volume 30 of *EPTCS*, 2010. 26

[MZ06] M. Odersky M. Zenger. Independently Extensible Solutions to the Expression Problem - Technical Report. Technical report, Ecole Polytechnique Federale de Lausanne (EPFL), Lausanne, Switzerland, 2006. 172

[NJ13] Rocco De Nicola and Christine Julien, editors. *Coordination Models and Languages, 15th International Conference, COORDINATION 2013, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec 2013, Florence, Italy, June 3-5, 2013. Proceedings*, volume 7890 of *Lecture Notes in Computer Science*. Springer, 2013. 26

[OD01] A. Omicini and E. Denti. From Tuple Spaces to Tuple Centres. *Science of Computer Programming*, 41(3):277–294, November 2001. 28, 34, 36, 40

[OZ] A. Omicini and F. Zambonelli. TuCSoN: a coordination model for mobile information agents. In Monica Divitini David G. Schwartz and Terje Brasethvik, editors, *1st International Workshop on Innovative Internet Information Systems (IIIS'98)*, pages 177–187, Pisa, Italy, 8–9 June. IDI – NTNU, Trondheim (Norway). 34, 40

[PA98] G.A. Papadopoulos and F. Arbab. Coordination Models and Languages. In *Technical Report SEN-R9834*. Centrum voor Wiskunde en Informatica (CWI), ISSN 1386-369X, 1998. 24, 26, 40

[Pan02] B. Panda, editor. *17th ACM Symposium on Applied Computing (SAC 2002)*, Madrid, Spain, 10–14 March 2002. ACM Press. Special Track on Coordination Models, Languages and Applications. 26

[PHW05a] A. Di Pierro, C. Hankin, and H. Wiklicky. Probabilistic Linda-based Coordiantion Languages. In F. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects (FMCO 2004)*, volume 3657 of *Lecture Notes in Computer Science*, pages 120–140. Springer, 2005. 40

[PHW05b] A. Di Pierro, C. Hankin, and H.b Wiklicky. Probabilistic Linda-based Coordination Languages. In *Proceedings of the Third International Conference on Formal Methods for Components and Objects*, FMCO'04, pages 120–140, Berlin, Heidelberg, 2005. Springer-Verlag. 39

[Plo81] G. Plotkin. A Structured Approach to Operational Semantics. (Computer Science Department, Aarhus University, DAIMI FN-19), 1981. 45

[Pro11] J. Proenç. *Synchronous Coordination of Distributed Components*. PhD thesis, Institute for Programming research and Algorithmics, University Leiden, Leiden, Netherland, 2011. 30, 32, 40

[RDP98] G.L. Ferrari R. De Nicola and R. Pugliese. KLAIM: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering (Special Issue on Mobility and Network Aware Computing)*, 1998. 28, 36, 40

[Sha92] E.Y. Shapiro. Embeddings Among Concurrent Programming Languages. In W.R. Cleaveland, editor, *Proceedings of COORDINATION 1992*, Lecture Notes in Computer Science, pages 486–503. Springer, 1992. 26, 49

[Sir12] Marjan Sirjani, editor. *Coordination Models and Languages - 14th International Conference, COORDINATION 2012, Stockholm, Sweden, June 14-15, 2012. Proceedings*, volume 7274 of *Lecture Notes in Computer Science*. Springer, 2012. 26

[SM113] *SAC 2013: Proceedings of the 28th Annual ACM Symposium on Applied Computing*, New York, NY, USA, 2013. ACM. Special Track on Coordination Models, Languages and Applications. 26

[Tuc04] TuCSoN Guide, tucson version: 1.4.0. http://lia.deis.unibo.it/Research/TuCSoN/doc/tucson.pdf, 2004. 34, 35

[Van09] O. Vandorpe. Modélisation de l'exécution de programmes dans les langages de coordination au moyen de réseaux de petri. Master's thesis, Faculty of Computer Science, University of Namur, Namur, Belgium, 2009. 287, 288, 291

[VC09] M. Viroli and M. Casadei. Biochemical Tuple Spaces for Self-organising Coordination. In J. Field and V. T. Vasconcelos, editors, *Proceedings of 11th International Conference, COORDINATION 2009*, volume 5521 of *Lecture Notes in Computer Science*, pages 143–162. Springer, 2009. 37, 38, 40, 82

[VC10] M. Viroli and M. Casadei. Chemical-Inspired Self-Composition of Competing Services. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2029–2036, New York, NY, USA, 2010. ACM. 37, 40

[WC115] *SAC 2015: Proceedings of the 30th Annual ACM Symposium on Applied Computing*, New York, NY, USA, 2015. ACM. Special Track on Coordination Models, Languages and Applications. 26

[Wyc98] P. Wyckoff. T Spaces. *IBM Systems Journal*, 37(3), 1998. 28

[Zav98a] G. Zavattaro. On the incomparability of Gamma and Linda. *Electronic Transactions on Numerical Analysis*, 1998. 82, 128

[Zav98b] Gianluigi Zavattaro. Towards a Hierarchy of Negative Test Operators for Generative Communication. *Electronic Notes in Theoretical Computer Science*, 16:154–170, 1998. 82, 128

List of Figures

List of Tables